VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Clusterovací platforma Terracotta

Clustering Platform for Java Application

2010

Bc. Dušan Kaloč

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6. května 2010 .....

Rád bych na tomto místě poděkoval všem, kteří mne při psaní této diplomové práce podporovali, zejména pak vedoucímu diplomové práce Ing. Štěpánu Kuchařovi za cenné rady a připomínky.

Abstrakt

Práce je věnována technologii Terracotta, která programátorům poskytuje zcela ojedinělý přístup k tvorbě distribuovaných systémů. Nepopisuje však pouze Terracottu samotnou, ale obsahuje také úvod do distribuovaných systémů. Poznatky z tohoto úvodu jsou pak využívány napříč celou prací, jež se dále věnuje historii Terracotty, jejímu teoretickému základu a způsobu, jakým se distribuované systémy nad ní implementují. V druhé části je pak popsána a navržena ukázková distribuovaná aplikace, která je poté realizována jak nad Terracottou, tak pomocí jiného frameworku. Obě tato řešení jsou také následně porovnána. Závěr práce se zabývá obecným převodem již existujících distribuovaných aplikací na platformu Terracotta.

Klíčová slova: Terracotta, Master/Worker, distribuované systémy

Abstract

This thesis is devoted to Terracotta which is a technology offering programmers an absolutely unique approach to developing distributed systems. It does not address only Terracotta though, but also contains an introduction to distributed systems. The findings from this introduction are then used throughout all the thesis, which further addresses the history of Terracotta, its theoretical fundamentals and with means how distributed systems are built upon it. In the second part there is given a vision and design of an example application which is then realized both upon Terracotta and by means of another framework. Both these solutions are then compared too. The final chapter deals with a guidance how already existing distributed applications can be ported to the Terracotta platform.

Keywords: Terracotta, Master/Worker, distributed systems

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
AWT	–	Abstract Window Toolkit
CD	–	Compact Disc
CPU	–	Central Processing Unit
DMI	–	Distributed Method Invocation
EDT	–	Event dispatching thread
EJB	–	Enterprise JavaBeans
GC	–	Garbage collection
GUI	–	Graphical user interface
HPFS	–	High Performance File System
HTML	–	HyperText Markup Language
IP	–	Internet Protocol
ISO	–	International Organization for Standardization
JAR	–	Java ARchive
JavaSE	–	Java Platform, Standard Edition
JDBC	–	Java Database Connectivity
JFC	–	Java Foundation Classes
JMS	–	Java Message Service
JMX	–	Java Management Extensions
JSF	–	JavaServer Faces
JVM	–	Java Virtual Machine
LFU	–	Least Frequently Used
LRU	–	Least Recently Used
NAS	–	Network-attached storage

NFS	–	Network File System
NTFS	–	New Technology File System
ORM	–	Object-relational mapping
OSI	–	Open System Interconnection
POJO	–	Plain Old Java Object
RMI	–	Remote Method Invocation
SQL	–	Structured Query Language
TCP	–	Transmission Control Protocol
UI	–	User interface
ZFS	–	Zettabyte File System

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Každý moderní člověk dnes používá internet. Vlastní email, sdílí soubory, hraje online hry, prohlíží si obrázky a videa a je členem mnoha sociálních sítí. Poněvadž je ale takovýchto lidí na světě hodně, musí se provozovatelé těchto služeb vypořádávat s enormní zátěží, kterou tito lidé na tyto služby vytvářejí každou vteřinu. Jak je tedy ale možné, že i když nakupuji na portálu eBay souběžně s dalšími desetitisíci uživatelů, eBay mi téměř ihned odpoví na každý můj požadavek a navíc se nikdy nestane, že by mne odmítnul z důvodu dosažení maximálního počtu uživatelů, který je aktuálně povolen. Žádná služba, a tedy ani eBay, by se proto nestala tím čím dnes je, pokud by uživatel na odpovědi čekal příliš dlouho, anebo by co chvíli dostával informaci, že aplikaci aktuálně používá příliš mnoho uživatelů a on tak musí bohužel počkat.

Cílem každé (celosvětové internetové) aplikace je tak vykazovat malé zpoždění (low latency) a vysokou propustnost (high throughput). První vlastnost znamená malou čekací dobu na odpovědi a ta druhá schopnost obsloužit co nejvíce uživatelů současně. Aby služba těchto vlastností mohla vůbec dosáhnout, nutným předpokladem je vlastnit výkonný hardware, na kterém bude aplikace provozována. Je zřejmé, že pro její spolehlivý běh nepostačí obyčejný počítač, který mám doma já anebo Vy, ale bude nutné zakoupit buď superpočítač, nebo mnoho (osobních) počítačů spojených prostřednictvím počítačové sítě. I přesto, že návrh aplikace pro superpočítač je vzhledem k druhé variantě o mnoho snazší, se superpočítači se dnes prakticky nesetkáme, neboť jsou extrémně drahé. Všechny služby, které známe a používáme (YouTube, Facebook, Twitter, Flickr apod.) jsou tak téměř se stoprocentní jistotou realizovány právě pomocí mnoha počítačů, které spolu komunikují prostřednictvím počítačové sítě. Těmto aplikacím, čili systémům složených z mnoha počítačů vzájemně komunikujících pomocí počítačové sítě, se obecně říká distribuované systémy nebo „klastry“[1] (z anglického slova „cluster“). Právě distribuovaným systémům, respektive Terracottě, což je specializovaný framework pro jejich vytváření, se věnuje tato práce.

Ve druhé kapitole vás tedy nejprve seznámím s problematikou tvorby distribuovaných systémů a ukážu, co je pro programátory tak obtížné při jejich návrhu a následné realizaci. S těmito znalostmi se poté pustím do popisu historie Terracotty a objasním, co vedlo k jejímu vzniku. Ve třetí kapitole uvedu její definici, podobnosti s NAS a vysvětlím, proč právě analogie k NAS je pro programátora i operátora distribuovaného systému výhodná.

Následující kapitoly budou již orientovány ryze prakticky. Nejprve tedy, na ukázkové aplikaci, ukážu, jak Terracotta vlastně funguje, a jak tuto aplikaci, pomocí této technologie, distribuovat. Z teoretických kapitol budete také vědět, že Terracotta nemá žádné API, a že tak jediným prostředkem, jak s ní komunikovat, je konfigurační soubor *tc-config.xml*. Po ukázkové aplikaci tak ihned uvedu výčet, téměř všech, jeho elementů a v závěru kapitoly vysvětlím, jak si aplikaci nad Terracottou vůbec spustit.

[1] Označení jakéhokoliv distribuovaného systému termínem klastr není zcela přesné, neboť klastrem se zpravidla myslí výpočetní systém používající pouze LAN a mající počítače se stejným hardware a operačním systémem.

V další části uvedu vizi a návrh reálně použitelné aplikace, kterou poté realizuji jak s Terracottou, tak bez ní. Obě tato řešení nakonec porovnám a to včetně výkonnostních parametrů.

V poslední kapitole této práce uvedu obecná pravidla pro převod libovolné distribuované aplikace sdílející stav na platformu Terracotta.

Napříč celou prací bylo čerpáno z [1].

2 Problematika distribuovaných systémů

Počátek distribuovaných systémů se datuje do druhé poloviny 80. let minulého století, kdy byly vynalezeny osobní počítače a vysokorychlostní počítačové sítě. Velmi důležitou vlastností každého distribuovaného systému je tzv. „transparentnost“. Tuto vlastnost se pokusím vysvětlit na službě YouTube, což je velmi oblíbená platforma pro sdílení videí. Já, jako uživatel, tuto službu (respektive její veřejné rozhraní) používám ke všemu, co mi tato služba nabízí, a nejenže tedy pomocí ní videa sleduji, ale také je v ní vyhledávám apod. Vnitřní organizace služby YouTube, čili to, z kolika a z jakých počítačů se skládá, je tedy přede mnou prostřednictvím jejího rozhraní skryto a já tedy nevím, zda počítač, který pro mne požadovaná videa vyhledává, je jiný než ten, který mi tato videa „streamuje“. Právě transparentnost dala vznik definici distribuovaného systému, viz definice 2.1.

Definice 2.1 *Distribuovaný systém je množina nezávislých počítačů, která se uživatelům jeví jako jeden souvislý systém [2].*

Distribuovaný systém se tedy skládá z mnoha počítačů a je zřejmé, že docílení malého zpoždění a vysoké propustnosti dosahuje pomocí co možná nejefektivnějšího využití dostupného výpočetního výkonu. To, jak je systém schopen tyto výpočetní zdroje využívat, se označuje jako tzv. škálovatelnost[2]. Platí, že pokud systém dobře škáluje, vykazuje také malé zpoždění a vysokou propustnost. Škálovatelnost je tedy další a zřejmě tou nejdůležitější vlastností distribuovaného systému a často se na ni ještě budu v dalším textu odkazovat, neboť Terracotta, čili technologie, jíž je věnována tato práce, přináší unikátní řešení, jak snadno jí lze dosáhnout.

V kontextu každé enterprise aplikace nebo internetové služby se vyskytují dvě zásadní role – vývojář a operátor. Úkolem vývojáře je vytvořit funkční aplikaci dle zadané specifikace a zodpovědností operátora tuto aplikaci spravovat po jejím nasazení. Oba dva se snaží zajistit, aby aplikace byla dostatečně škálovatelná a zvládla tak reálnou zátěž.

Škálovatelnost systému je především dána jeho architekturou, která určuje programovací model a míru, s jakou se na výsledné škálovatelnosti podílí operátor, respektive vývojář. Platí, že čím méně ovlivňuje výslednou škálovatelnost vývojář a čím více operátor, tím lépe. Ihned vysvětlím proč. Tím důvodem je počítačová síť, která, jak už víme, je pro běh distribuovaného systému nezbytná. Neboť je ovšem ze své podstaty nespolehlivá, nezabezpečená a vykazuje vysoké zpoždění a omezenou propustnost, je to právě ona, respektive tyto její vlastnosti, které činí návrh distribuovaného systému obtížným. Se všemi těmito nedostatky, čili pomalostí a nespolehlivostí sítě apod. musí vývojář při návrhu distribuovaného systému počítat a je zřejmé, že čím více komunikační logiky realizuje zvolená architektura za něj, tím lépe.

Cílem je tedy osvobodit vývojáře, respektive jeho zdrojový kód, od řízení, co a kdy se po síti posílá a ponechat toto čistě na architektuře systému. Nejenom, že se zbavíme chyb, které se vývojáři při realizace komunikace často dopouští, ale tím, že komunikační logika

[2]Škálovatelností je několik typů, ale v textu této práce je škálovatelnost chápána především ve smyslu schopnosti systému zvýšit (či snížit) své zdroje tak, aby byla co nejefektivněji zpracovávána aktuální zátěž.

již není součástí aplikačního kódu, mohou být všechny optimalizace vedoucí ke zvýšení škálovatelnosti prováděny pouze operátorem a to technikami, které on dobře zná.

Tou klíčovou vlastností opravdu škálovatelného systému je tedy transparentnost jeho architektury. Nemyslíme zde transparentnost distribuovaného systému jako takového, ovšem transparentnost ve smyslu, že vývojář neví a ani ho to vlastně nezajímá, jak a s jakými počítači distribuovaného systému právě komunikuje.

Nejpoužívanější a nejznámější transparentní architekturu poskytují databáze, a pokud pominu specializované typy distribuovaných systémů jako výpočetní systémy apod., tak databáze byly po více než třicet let jedinou možností, jak dosáhnout opravdu škálovatelného distribuovaného informačního systému.

Typickými systémy, které používají databázi, jsou webové aplikace. Tyto aplikace obvykle používají tzv. „třívrstvou architekturu“ (viz obrázek 1), jejíž první vrstva realizuje styk s uživatelem a nazývá se „*prezentační*“. Je tvořena webovými servery, které nám do našich prohlížečů zasílají HTML. První vrstva tedy dává vzhled informacím, které já jako uživatel požaduji, ale tyto informace sama nevytváří. Toto je totiž úkolem druhé, tzv. „*aplikační vrstvy*“, jejíž aplikační servery realizují vlastní business logiku. Data, se kterými aplikace pracuje, jsou uloženy v databázi, která představuje třetí a konečně poslední vrstvu této vrstvené architektury (vrstva je nazývána „*datová*“).

Vlastní aplikaci distribuovaného systému tedy realizuje druhá, aplikační vrstva. Komunikace zde mezi počítači ovšem neprobíhá přímo, nýbrž pouze prostřednictvím databáze a transakcí. Transparentnost tedy tkví v tom, že jednotlivé aplikační servery o sobě vůbec nic neví a vývojář tak nemusí řešit jejich vzájemnou komunikaci. Návrh distribuovaného systému tak spočívá v pouhém efektivním využití jazyka SQL, transakcí a uložených procedur.

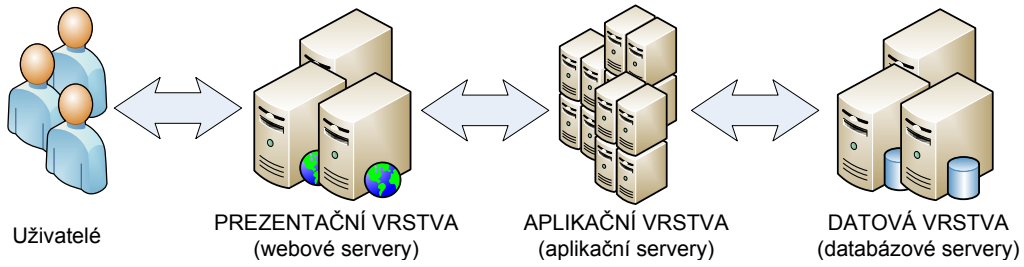
Škálovatelnost systému tak neurčuje druhá, nýbrž třetí vrstva systému. Neškálujeme tedy aplikaci jako takovou, ale škálujeme pouze vlastní databázi. Na třetí úrovni operuje pouze operátor a škálovatelnosti tak dosahuje pomocí jemu známých technik jako distribuce, fragmentace a replikace dat. Operátor je tak zcela nezávislý jak na aplikační vrstvě, tak na vývojářích a jakákoliv optimalizace vedoucí k růstu škálovatelnosti neznamená nutnou změnu zdrojového kódu, jak je tomu často u jiných řešení[3].

Za svůj úspěch na poli informačních systémů tak databáze vděčí především svému programovacímu modelu (SQL, transakce, popřípadě ORM[4]) a existenci obrovského počtu nástrojů pro jejich správu. Jazyk SQL, uložené procedury apod. zná dnes prakticky každý programátor a také nástroje pro správu jsou každodenně používány pro řízení obrovských distribuovaných systémů.

Zdá se tedy, že použití databáze je opravdu velmi výhodné, neboť škálovatelnosti dosahujeme jednoduchým programovacím modelem a nezávislou správou. Pokud by tomu tak opravdu bylo a databáze by byly opravdu bezchybné, jednoduše by ale nebylo potřeba technologií jako je Terracotta. Není tomu tedy tak, a jak v následujícím textu ukážu, i databáze má, bohužel, své stinné stránky.

[3]U jiných řešení (avšak mimo Terracotty, jak popíši později), musí programátor komunikaci mezi jednotlivými počítači distribuovaného systému řešit sám, čili ve svém zdrojovém kódu využít například „sokety“. Více informací o „soketech“ je uvedeno například zde: http://en.wikipedia.org/wiki/Internet_socket.

[4]Více informací viz http://en.wikipedia.org/wiki/Object-relational_mapping.



Obrázek 1: Třívrstvá architektura

Stále více dnešních databázových aplikací nepracuje již se surovými relačními daty, ale pouze s objekty. Databáze ovšem ze své podstaty práci s objekty neumožňují, a proto aplikace pro převod relačních dat na objekty používají techniku zvanou ORM. Poněvadž je ale tato technika velmi oblíbená, je zřejmé, že vývojáři by ihned raději pracovali pouze s objekty a ne s relacemi či tímto ORM. Jelikož ale čistě objektové technologie nenabízejí pro tvorbu distribuovaných systémů to, co databáze, je pro aplikace bohužel ORM často jedinou volbou. Ukazuje se tedy, že i přes jednoduchý programovací model je relační podstata databází zároveň jejich velkou nevýhodou. Chybí zde tedy ta svoboda v návrhu, kdy vývojář je nucen přizpůsobit se tomuto modelu a nemůže si aplikaci navrhnout tak, jak chce, tedy ryze objektovým způsobem.

Definujeme-li lineární škálovatelnost[5] jako stoprocentní využití veškerého dostupného výpočetního hardware, pak další nevýhodou databází je, že tuto škálovatelnost nenabízejí. Ihned vysvětlím proč. Výpočetním hardwarem u třívrstvých aplikací rozumíme aplikační servery, které přistupují k databázi. Poněvadž ale databáze čte data z pevného disku, je zpoždění vrácení odpovědi relativně velké. V okamžiku, kdy tedy aplikační server čeká na odpověď databáze, zůstává jeho výpočetní výkon nevyužit. Tato skutečnost vede samozřejmě ke snížení celkové škálovatelnosti a z tohoto důvodu platí, že databázové systémy zpravidla neškálují lineárně.

2.1 Přístupy ke škálování

Předchozí text nás letmo zavedl do prostředí distribuovaných systémů, kde zásadní roli hrají databáze. Seznámili jsme se s jejími výhodami, ale také nedostatky, a i přesto, že pomocí technik jako je replikace a fragmentace dat můžeme dosáhnout dostatečné škálovatelnosti, může se i databáze stát vážným výkonnostním problémem. Ke snazšímu pochopení, proč tomu tak může být, slouží právě tato podkapitola.

V minulosti byly aplikace jen vzácně rozprostřeny na více počítačů. Přistupovalo se pouze k databázi, která navíc obsahovala mnoho naší aplikační logiky. Toto se ovšem radikálně změnilo s příchodem internetu, respektive třívrstvé architektury, o které jsem se

[5]Definic lineární škálovatelnosti je více, ale v textu této práce je lineární škálovatelnost chápána především tak, že je daná aplikace schopna škálovat donekonečna, respektive že nikdy nedosáhne žádných bariér ve smyslu škálovatelnosti. Pěkný popis lineární škálovatelnosti naleznete například zde: http://www.gigaspaces.com/wiki/download/attachments/1835009/FromDeadEndToOpenRoad.pdf.

již zmiňoval. Uživatel již nechtěl pouhé statické HTML stránky, ale vyžadoval dynamický obsah. Ryze webové aplikace se tak náhle staly také podnikovými.

Pro společnosti se však aplikační vrstva ukázala být vážným problémem, neboť s internetem najednou přicházelo o mnoho více požadavků a databáze tak jednoduše „nestíhala“. Vzhledem k tomu, že potřebnou škálu často odhalíme až v době po nasazení, ukázaly se navíc správa a deployment databázové a aplikační vrstvy jako drahé a neefektivní.

I přes všechny tyto problémy se však třívrstvá architektura stávala novým, používaným a akceptovaným modelem. K tomu jí pomohla především Java, kterou si mnozí v té době zvolili pro její flexibilitu a zároveň jednoduchost.

Preferovaný model nasazení byl ten, který mnozí nazývali jako tzv. „scaling-out“. Ten označuje nám již známý model mnoha počítačů a je protikladem ke „scaling-up“ neboli modelu jednoho (super)počítače. Poslední silou, která již tak natrvalo ukotvila pozici třívrstvé architektury, byl Linux. Bez něj, respektive open-source, by nám jinak vysoké licenční poplatky nedovolily používat tolik aplikačních serverů a ostatních počítačů. Ve výsledku si tak většina architektů zvolila scaling-out, namísto scaling-up.

Společnost Oracle, jako největší databázový výrobce, se tehdy nijak neobával příchodu třívrstvé architektury. Internet v té době nebyl ještě příliš rozšířen, a tak ani zátěž generovaná na tuto architekturu, respektive databázi, nebyla nijak ohromná. Databáze tudíž na tehdejší zatížení stačily, a proto v té době jen málokdo tušil, že vlivem blížícího se internetového boomu (respektive enormního nárůstu uživatelů, a tedy i celkové zátěže), tomu tak již brzy nebude.

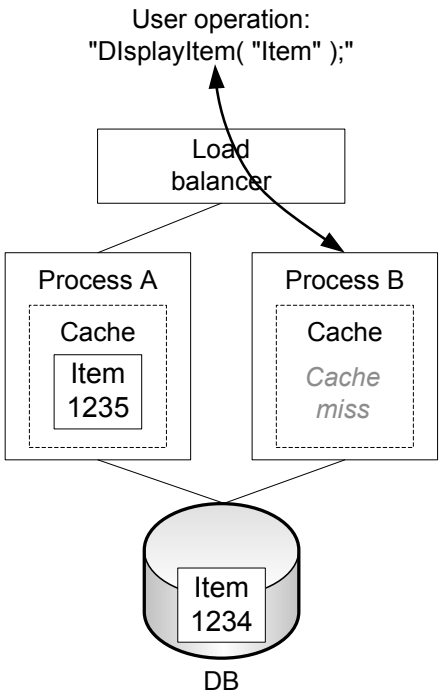
Zanedlouho tedy nastala tahanice o to, jak škálovat. Na jedné straně stálo složité a drahé „klastrování“ a na straně druhé jednoduché, v úvodu již popisované, aplikace s databázovým „backendem“. Popis těchto přístupů, který je uveden dále, nám pomůže si uvědomit, že aplikace, která se pokouší škálovat, může vlastně používat stejné techniky jako architektura víceprocesorové základní desky. Blíže vás s tímto konceptem seznámím v následující kapitole a ukážu vám, jak tento nový koncept dělá škálování transparentní a jak nám tato transparentnost pomáhá v aplikačním návrhu.

Základní výzva ve škálování spočívá v paralelním přístupu mnoha procesů k aplikačním datům. Tento souběžný přístup musí být samozřejmě určitým způsobem zabezpečen a v kontextu webových aplikací toho docílíme buď tzv. „stavovým“, či „bezstavovým“ přístupem. Ani jeden z nich se však bohužel nedokáže vyhnout výkonnostním problémům, což ukážu v následujících odstavcích.

2.1.1 Škálování databáze

Jedna z možností, jak rozšířit aplikaci mezi více procesů, je zajistit, že žádný proces si neuchovává aplikační data v paměti. Tento přístup se označuje jako tzv. bezstavový (stateless) a typicky ho realizujeme zápisem všech dat do databáze.

V dobách, kdy aplikace existovaly pouze v jedné instanci, se pro omezení komunikace s databází používalo tzv. „cachování“. Urychlení spočívalo v tom, že si aplikační server, pro rychlejší přístup, uchovával určitá „business“ data v paměti. Obrázek 2 ovšem ukazuje problémy, jaké nám tento přístup přináší, pokud ho aplikujeme na aplikaci složenou z více



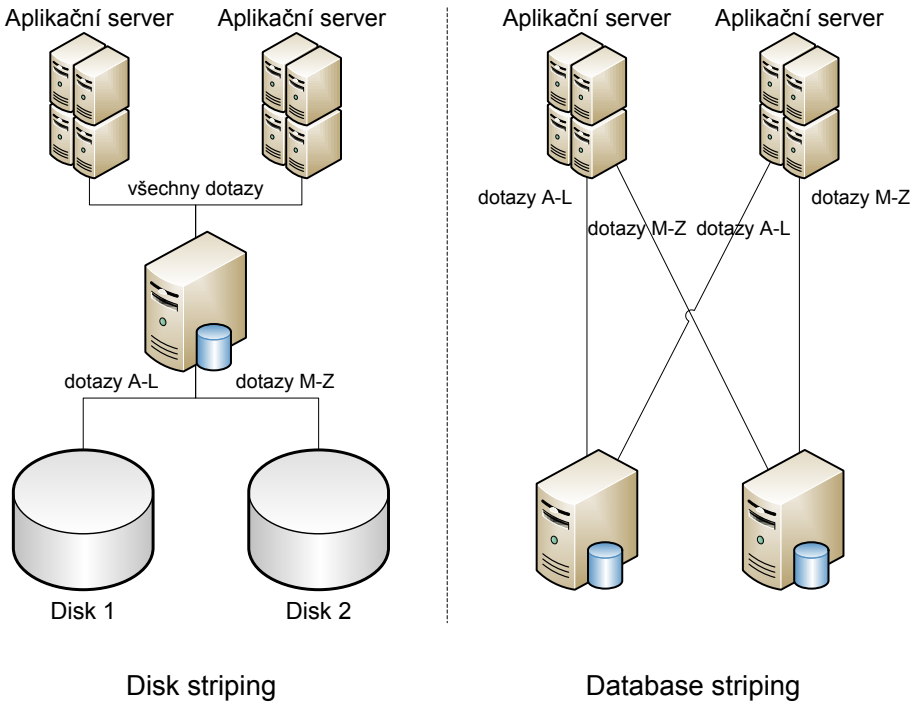
Obrázek 2: Nedistribuovaná cache vede k nekonzistencím [1]

procesů. Poněvadž zde má každý proces svou unikátní „cache“, může se stát, že jeden proces neuvidí změny toho druhého. Pokud by tedy proces *A* na obrázku 2 položku *Item* změnil a tuto změnu by ještě nezapsal do databáze, mohlo by se stát, že proces *B* vrátí koncovému uživateli stará data.

Řešením je tedy cache úplně odstranit, anebo distribuovat. Distribuce cache spočívá v existenci pouze jediné cache společné všem procesům, a poněvadž je na ní založena Terracotta, budu o ní mluvit později. Co nám ale přinese její odstranění? Bohužel pouze výkonnostní problémy, neboť díky procesům, které si neukládají žádná data v paměti, bude databáze zahlcena zbytečnými SQL dotazy. Zbytečnými z toho důvodu, že i ta data, se kterými již proces někdy pracoval, a která od té doby nebyla nijak ostatními procesy změněna, budou muset být i přesto načtena opětovně z databáze.

Vypnutím cache čili bezstavovým přístupem sice docílíme perzistence a konzistence, ale výsledný systém nám, díky redundantním dotazům, nebude dobře škálovat. Jediným řešením je tak škálování vlastní databáze, které je nám trochu známo již z úvodní kapitoly.

Koncept škálování databáze spočívá v tzv. „striping“ (neboli rozdělení) a jeho dvě varianty ukazuje obrázek 3. *Disk striping* pomáhá databázovému serveru zvládnout více aplikační zátěže rozprostřením dat na více pevných disků a je zřejmé, že jejich větší počet udělá více než jen jeden. Stále se ale nezbavíme těch redundantních dotazů, neboť databáze pořád existuje pouze v jedné instanci. Druhou variantou je proto striping vlastních databází (*database striping*), která ale zase pro změnu vyžaduje spolupráci vývojáře. Z obrázku 3 je totiž patrné, že je nyní na aplikaci, aby si pamatovala, ve které databázi jsou



Obrázek 3: Disk striping vs. database striping [1]

ta správná data uložena. Toto pamatování může jistě udělat i jenom samotná databáze, ovšem v tom případě by data té části, kterou ta daná instance nemá, musely poskytnout ostatní databáze. Databáze by tak musely mezi sebou komunikovat, což by ale ve výsledku nevedlo ke kýžené dvojnásobné kapacitě.

Bezstavový model tedy není tou nejlepší architekturou, jak by se na první pohled mohlo zdát. K docílení škálovatelnosti zde totiž často potřebujeme spolupráci vývojáře a/nebo složité striping. Alternativou je tak replikace dat v paměti, kterou popisuji v dalším odstavci.

2.1.2 Replikace dat v paměti

Replikace dat, stejně jako cachování, uchovává data v operační paměti. Je tedy opakem bezstavového modelu a z tohoto důvodu se tento přístup nazývá také stavovým (stateful). Na rozdíl od cachování si však data jednotlivé počítače neukládají nezávisle na ostatních, ale každý počítač pracuje pouze s jednou kopií (replikou) jedněch a těch samých dat. Replikací dat tak získáme nejenom vyšší dostupnost, ale z důvodu, že data jsou uložena v operační paměti, také téměř nulové zpoždění.

Výhoda replikace tedy spočívá v tom, že každý počítač teď vidí pouze jeden a tentýž stav. Aby to však bylo možné, repliky si musí být navzájem konzistentní, což představuje největší problém stavového modelu. A protože konzistence znamená, že jakákoliv změna stavu, provedená některým z počítačů, musí být ihned viděna také ostatními, vyžaduje

její udržování poměrně značnou komunikaci po síti. Poněvadž již ale víme, že síť je „pomalá“ a navíc aplikační servery nemohou po dobu synchronizace vykonávat žádnou další činnost, představuje tato tzv. „silná“ (též „přísná“) konzistence vážný výkonnostní problém. Abychom tomuto problému předešli, a využili tak opravdu výhody uložení dat v paměti, musíme buď tuto silnou konzistenci „obejít“, anebo využít jiný, volnější konzistentní model [6].

Zbavení se nutnosti zachovávat konzistenci můžeme i zde docílit pomocí nám již známého „stripingu“. Koncept je stále stejný, pouze s tím rozdílem, že nerozdělujeme již tabulku dat mezi více databází, ale pro změnu mezi více „pamětí“. Datovou tabulku tak například rozdělíme mezi dva servery, kde první půlka dat bude uložena na jednom a ta druhá na druhém. Poněvadž jsou ale tato data uložena v paměti, můžeme k nim přistupovat, v porovnání k bezstavovému modelu, téměř s nulovým zpožděním.

Poté, co se zdárně „popereme“ s konzistencí, musíme však bohužel čelit ještě dalšímu vážnému problému. Tím problémem je tzv. „dostupnost“, která znamená, že jakákoliv část sdílených dat musí být také zapsána na perzistentní médium. Dostupnost dat je důležitá z toho důvodu, že pokud by například v datovém centru, na kterém běží naše aplikace, vypadl proud, aplikace bude po jeho obnově schopna pokračovat tam, kde ve vykonávání před výpadkem skončila.

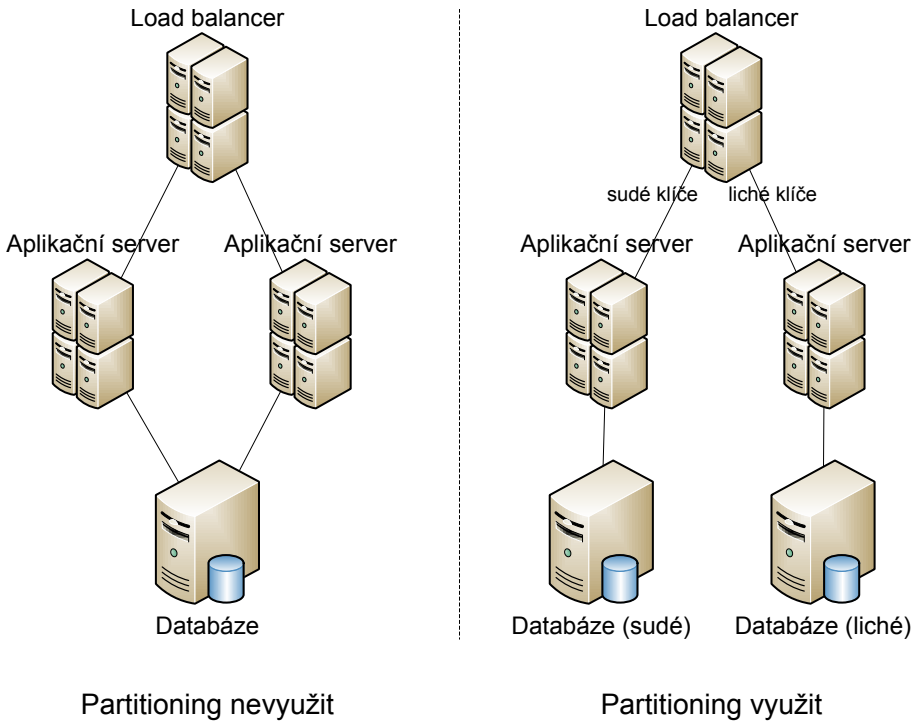
Poněvadž ve stavovém modelu pracujeme s kopiemi a navíc udržování jejich konzistence ve světě Javy znamená serializaci a deserializaci grafů objektů, je programovací model stavového přístupu pro vývojáře poměrně složitý.

I přesto, že stavový model uchovává aplikační data v paměti, není nárůst ve škálovatelnosti, díky jeho jiným problémům, oproti bezstavovému přístupu vždy zaručen. Život s replikami je navíc složitý a můžeme tak tedy říci, že ani stavový přístup nepřináší ten ideální model pro realizaci distribuovaného systému.

2.1.3 Partitioning aplikačních dat

Replikace versus „partitioning“. Veškerá analýza, kterou jsme doposud vykonali, se týkala tohoto paradigma. Partitioning je architektura, která snižuje riziko výkonnostních problémů, týkajících se přílišného přenosu po síti či nadbytečné komunikace s databází, rozdělením dat a odpovídající práce napříč klastrem serverů. Tento koncept ilustruje obrázek 4, kde typická databázová aplikace získává dvojnásobnou databázovou kapacitu rozdělením klíčů na sudé a liché. „Partitioning“ je tedy proces, při kterém se data rozdělí na více nezávislých celků, ke kterým poté můžeme přistupovat nezávisle. Rozdíl mezi stripingem a partitioningem je tedy takový, že u partitioningu aplikační server, respektive aplikační logika neví, že data jsou rozdělena na více částí. O tomto rozdělení ví pouze vyšší úroveň (na obrázku 4 reprezentována load balancerem) a ta vždy určí, jakému serveru předá přijatý požadavek. Rozhodne se tedy na základě toho, jaké části dat je požadavek určen. U stripingu jsou data také rozdělena na více částí a každá část je opět

[6] Více o konzistentních modelech se můžete dočíst například zde: http://en.wikipedia.org/wiki/Consistency_model.



Obrázek 4: Partitioning [1]

uložena na jiném fyzickém úložišti, ale o tomto rozdělení musí vědět již aplikační logika, neboť přerozdělování požadavků již neprovádí vyšší úroveň, ale ona samotná.

Poněvadž když aplikace roste, replikace dat nebo její správa se stává čím dál víc složitou, a tak pro mnohé aplikační týmy se stal partitioning jedinou schůdnou cestou k opravdové škálovatelnosti. Jak již víme, aplikace se stává rychlejší buď cachováním, a uchováváním dat tak v paměti, nebo jejich „stripingem“ mezi více počítačů. Jinými slovy, dostupnosti nejlépe dosáhneme bezstavovým modelem, zatímco škálování nás nutí umisťovat data v paměti a replikovat. Ve skutečnosti jsou však aplikace stavové a bezstavové zároveň a to je to, co si Terracotta uvědomila. Vývojáři a operátoři tak potřebují konečně způsob, jak překonat ten kompromis mezi dostupností a škálovatelností, aniž by se museli vydat předčasně cestou „partitioningu“.

2.2 Shrnutí

Cílem každé internetové služby je vykazovat co nejmenší zpoždění a co nejvyšší propustnost. S ohledem na nízkou cenu osobních počítačů a existenci vysokorychlostních sítí toho dnes služby dosahují pomocí distribuovaných systémů. Ta obrovská reálná zátěž je tak rozdělena na mnoho počítačů, které spolu komunikují prostřednictvím počítačové sítě.

Právě tato nutná komunikace však činí návrh distribuovaného systému značně obtížným, neboť počítačová síť přináší mnohá úskalí. Každý vývojář musí především počítat,

že síť vykazuje určité zpoždění a omezenou propustnost, což nám ke kýžené škálovatelnosti samozřejmě nepřidá. Poněvadž je tedy programování komunikační logiky náročné, často chceme, aby toto za nás realizovala architektura. Jelikož je tedy škálovatelnost dána především efektivní komunikací, čím méně se na ní bude podílet vývojář a čím více operátor, tím lépe.

Dlouholetá praxe ukázala, že velmi výhodnou architekturu nabízejí databáze. O škálovatelnost se zde stará výhradně operátor a zjednodušeně řečeno, vývojář pouze čte a zapisuje data. Neexistuje zde tedy komunikace jako taková, ale pouze ta, prostřednictvím transakcí a dat. Poněvadž škálovatelnost tedy nesouvisí vůbec se zdrojovým kódem aplikace, pro její růst ho není třeba nijak měnit a operátor tak může používat pouze techniky jako fragmentaci a replikaci dat. Problémy databáze ovšem tkví v její relační podstatě a stálém přístupu na pevný disk. Tyto, a jak dále uvidíme, i další nedostatky eliminuje Terracotta, která přináší unikátní řešení, jak se databáze zbavit a zároveň zachovat všechny její dobré vlastnosti.

3 Historie Terracotty

Se znalostmi o cachování, replikaci a partitioningu se nyní můžeme pustit do historie Terracotty a analýzy důvodů, proč vůbec vznikla.

Počátek Terracotty sahá do roku 2000, kdy její tvůrci vytvářeli e-commerce aplikaci, která se zanedlouho stala členem „top ten“ na světě. Aplikace měla být stavová a její workflow mělo být řízeno transakcemi v databázi. Právě ale tyto transakce se záhy ukázaly jako největší chyba v návrhu a ta rozhodnutí, která se původně zdála být o mnoho důležitější, se neukázala jako zas až tak kritická.

Autoři chtěli mít původně celý workflow neboli konverzační stav uložen v paměti, avšak v té době neexistovalo žádné čistě „in-memory“ řešení, které by škálovalo lineárně a zároveň by replikovanému stavu zajistilo vysokou dostupnost. Poněvadž riziko ztráty konverzačního stavu, uchovaného jenom v paměti, bylo veliké, skončili tvůrci Terracotty u stateful/stateless hybridu. Veškerá databázová data tak byla, pro rychlý přístup, cachována a replikována v paměti, ovšem zápis byl pro zaručení dostupnosti veden ihned do databáze.

Alternativou k této architektuře byly distribuované transakce, ovšem žádné řešení okolo roku 2000 neposkytovalo tuto funkcionalitu na úrovni JVM. Realizace vlastního „in-memory“ distribuovaného systému s transakčními zárukami se zdála také být příliš složitá a poslední alternativou tak zůstala tzv. „eventuálně konzistentní architektura“[7]. Tu však autoři rychle zavrhli a ponechali si raději původní hybridní řešení.

3.1 Aplikace je jeden velký počítač

Už za pár týdnů, po prvotním spuštění systému, se ukázalo, že aplikace svou reálnou zátěž nezvládá. Tím viníkem byla databáze a autoři Terracotty si tak zanedlouho uvědomili: na třívrstvou architekturu bychom měli nahlížet, jako kdyby byla jeden obrovský multiprocesorový počítač.

K efektivnímu běhu procesoru se využívá vrstvená architektura pamětí. Poněvadž je pevný disk, respektive hlavní operační paměť počítače pomalá, jsou mezi nimi a procesorem umístěny L1, respektive L2 cache. Cache tak vyrovnávají ty ohromné rozdíly v rychlostech, které mezi nimi, čili procesorem, pamětí a pevným diskem, jsou. Aby se tedy procesor nezpomaloval čekáním na data z hlavní operační paměti, čte je pouze z L1, která pro změnu cachuje data z L2 a až ta přistupuje nakonec k hlavní paměti. L1 je tak nejrychlejší a čtení z ní tak procesor téměř nezbrzdí. L2 je naopak řádově pomalejší, ale čtení z ní je stále o mnoho výhodnější, než kdybychom ihned přistupovali k operační paměti. L1 vždy patří pouze jednomu jádru nebo jednojádrovému procesoru, naopak L2 je mezi jádry, respektive více procesory sdílena. Toto sdílení přináší totiž ohromné zrychlení při přepínání kontextu, kdy je potřeba přenést stav určitého vlákna z jedné L1 do druhé a L2 nám tak zajistí, že tento stav nemusíme dočasně ukládat do pomalé operační paměti. I když je přepínání kontextu, v porovnání s rychlostí L1, i tak stále relativně pomalé, nárůst v rychlosti, oproti jinak nutnému přístupu do hlavní paměti, je i přesto ohromný.

[7] Více viz tato adresa: http://en.wikipedia.org/wiki/Eventual_consistency.

Poněvadž aplikaci přístup na svou vlastní haldu téměř nic nestojí, tak v analogii mezi architekturou počítače a aplikací, můžeme L1 chápat jako cachování v rámci JVM. Stejně jako třívrstvá aplikace stráví čekáním na odpověď databáze v průměru několik stovek milisekund, tak i CPU ztratí několik tisíc cyklů čekáním na hlavní paměť nebo pevný disk. Přístup k databázi je tak drahý, stejně jako v případě operační paměti počítače.

Poté, co si toto tvůrci Terracotty uvědomili, instalovali cache do každého Java procesu. Dále, jako nadmnožinu všech L1, vytvořili L2, kterou umístili mezi databázi a tyto jednotlivé procesy. Aby však tato nová architektura přinesla užitek, muselo být volání L2 (ideálně řádově) rychlejší, než přístup k databázi v původním hybridním řešení.

Potom, co se toto povedlo realizovat, dokázal autorům, tento zcela ojedinělý přístup, eliminovat několik výkonnostních problémů. Jejich e-commerce aplikace tak ve výsledku začala konečně škálovat a to díky těmto důsledkům nové L2 cache:

- Jednotlivé procesy již nemusí čekat na databázi
  - A protože je L2 oproti databázi řádově rychlejší, výsledný L1/L2 systém tak (téměř) lineárně škáluje.
- Radikálně snižuje cenu přenosu úkolů mezi jednotlivými aplikačními servery.
- Aplikační servery mohou zůstat velmi „malé“
  - Aplikační procesy již totiž nemusí cachovat všechna data u sebe v L1, ale mohou je, bez jakýchkoliv problémů, ponechat pouze v L2
  - Bez L2 totiž docházelo ke snížení celkové škálovatelnosti, neboť to, co se do paměti aplikačního serveru již nevešlo, muselo být uloženo přímo v databázi. Abychom tomuto předešli, aplikační servery musely být „velké“, neboťmusely mít mnoho operační paměti.
- Nekompromisně omezuje komunikaci s databází
  - Díky L2 je počet dotazů na databázi snížen jen na opravdu nutné minimum a představuje tak jen zlomek toho, co aktuálně na aplikaci generují všichni její koncoví uživatelé.
- Poskytuje vyšší dostupnost
  - Data jsou totiž k dispozici, i když je databáze zrovna mimo provoz.

Díky L2 konceptu tak e-commerce aplikace získala jednoduchou škálovatelnost a zároveň vysokou dostupnost. S L2 se tak role databáze v dosahování škálovatelnosti značně snížila a díky L1, uvnitř každého aplikačního serveru, je nyní dosažitelná dokonce i lineární škála.

Stejně jako tato aplikace, i architektura Terracotty používá L1 a L2 cache. I pro ni tudíž platí analogie s architekturou počítače, neboť obsahuje knihovny, které instalují L1 do jednotlivých aplikačních JVM transparentně a zároveň je její součástí server, který se chová úplně stejně jako L2.

3.2 Terracotta, aneb obecně použitelná L2

L2 se tedy ukázalo jako velmi výhodné řešení, jak se šikovně „zbavit“ databáze a získat tím jednoduchou škálovatelnost a zároveň zachovat vysokou dostupnost. Toto řešení ovšem stále nebylo perfektní a já se nyní pokusím vysvětlit proč.

3.2.1 Invazivnost původní L2

I přes to, že zakladatelé Terracotty, čili tvůrci diskutovaného e-commerce systému, vytvořili L2 opravdu efektivně a tato cache ve výsledku koncového uživatele téměř nezpomalovala, tak její koncepce přinášela stále mnoho problémů.

L2 fungovala na principu uchovávání párů klíč/hodnota, a kdykoliv tedy vývojář změnil stav sdíleného objektu, tak aby se o této změně mohly dozvědět i ostatní repliky, musel tyto úpravy nejdříve serializovat a následně aktualizovat příslušný klíč. Co vývojář tedy do cache nedal, to tam nebylo a tento přístup samozřejmě přinášel problémy:

- S konzistencí – pokud tedy vývojář zapomněl změny serializovat a následně replikovat do L2, konzistence replik byla ihned porušena, neboť ostatní aplikační servery jednoduše neměly možnost, jak se o těchto úpravách dozvědět
- Se škálovatelností – k výkonnostním problémům docházelo v případě, že programátor replikoval změny až příliš často a to tedy i tehdy, když to nebylo nutně potřeba. Také se stávalo, že změny vývojáři ukládali přímo do databáze a cachování, respektive L2 tak zcela opomněli

Podstata těchto problémů, čili nedostatků popisovaného konceptu L2, tkví v přítomnosti API. Abychom totiž využili toho, co nám API nabízí, musíme ho nejprve do svého kódu integrovat. A protože L2 slouží ke sdílení objektů, tak integrace jeho API spočívala ve vytvoření složité replikační logiky. Na tomto API tak neustále pracovala více než stovka programátorů, která, aby systém zvládal svou reálnou zátěž, přicházela se stále novými a výkonnějšími replikačními technikami a nástroji. Namísto toho, aby se tedy ve vývojových cyklech systému implementovala jeho nová funkcionalita, tak docházelo, za účelem optimalizací, pouze k refaktorizaci stávajícího kódu. Správa škálující aplikace, respektive jejího zdrojového kódu, se tak, díky těm nikdy nekončícím úpravám, ukázala jako příliš složitá a značně zpomalující nový vývoj.

Zkušenosti z e-commerce projektu tak bohužel jenom ukázaly, že stávají koncept L2 tu vysokou cenu za škálování pouze přesunul z drahého databázového hardware do složitého aplikačního vývoje. Ve výsledku tak cena za scaling-out aplikace byla stále příliš vysoká.

Jediným možným řešením tak bylo udělat L2 cache, respektive L2 server transparentní. Cílem zakladatelů Terracotty, čili Terracotty jako takové, tak bylo přinést bezstavové prostředí pro operátory, kde obsah L2 je automaticky ukládán na pevný disk a zároveň stavový programovací model pro programátory, kde L2, respektive L1 jsou pro ně, čili aplikaci, naprosto transparentní a kde problémy s replikací a konzistencí tak neexistují.

3.2.2 Transparentnost rovná se obecná použitelnost

V úvodu této kapitoly jsem říkal, že tvůrci Terracotty, na počátku jejich e-commerce projektu, dělali rozhodnutí, která měla mít, podle nich, na výsledný výkon produktu největší vliv. Přeli se tedy o to, zda zvolit Struts[8] či JSF[9], nebo jestli použít Tomcat[10] či raději jiný servletový kontejner. Tato debata se ovšem ukázala být, jak již víte, o mnoho méně důležitá, než ta, která pojednává o klíčovém modelu škálovatelnosti a dostupnosti.

Ze svých zkušeností autoři Terracotty ví, že i když vycházíme z bezstavového modelu, tak v důsledku optimalizací jeho zpoždění, dojdeme bohužel pouze k názoru, že jeho propustnost je příliš malá. Praxe ukázala, že všechna řešení tohoto problému však skončí u cachování, a že tedy ten klíčový model škálovatelnosti se tak bez něj neobejde. Cachování si ovšem vynucuje změnu doménového modelu, neboť v případě replikace je potřeba určitého konzistentního protokolu. Předcházející text ovšem ukázal, že cachování na úrovni zdrojového kódu vede bohužel ke složité replikační logice, kterou je ještě navíc, ve většině případů, potřeba neustále optimalizovat. Nejenže tak tyto, nikdy nekončící, optimalizace brání v novém vývoji, ale často jsou také ušity pouze pro konkrétní řešení. To byl také případ e-commerce systému, kde jeho programátoři vytvořili sice mnoho efektivního, avšak již ne obecně použitelného kódu.

Z těchto důvodů proto zakladatelé Terracotty usilovali o ryze transparentní řešení, které by automaticky pracovalo uvnitř JVM a nemělo by tak žádné API. Aplikace by tak mohly být zcela obecně použitelné, neboť tam kde není API, není ani potřeba žádného složitého replikačního kódu.

Původní řešení L2 tak bylo nahrazeno zcela transparentním L1/L2 systémem, kde aplikace, stejně jako v případě cache procesorů, ani neví, že ke cachování vůbec dochází. Cachování tak již není starostí programátorů, ale pouze L1, respektive L2, která je také tím novým centrálním prvkem, který se zde stará o koordinaci aplikačních aktivit.

3.2.3 Transparentní škálovatelnost s dostupností

Škálovatelnost a dostupnost jsou opačné síly. Jakýkoliv zápis na pevný disk nás zpomalí a cokoliv, co ponecháme zase jen v paměti, vystavujeme nebezpečí, že se v případě chyby procesu či výpadku proudu, ztratí. Paměť je také ta nejrychlejší část I/O počítače, zatímco disk ta nejpomalejší. Nejhorší ovšem je, že počítačová síť se bohužel blíží svou rychlostí spíše pevnému disku a ne operační paměti.

Jak již tedy víme, počítačová síť činí návrh distribuovaného systému složitým, neboť jeho jednotlivé počítače musí po ní komunikovat. Je tedy zřejmé, že čím méně systém po síti komunikuje, respektive čím méně toho po ní posílá, tím lépe.

Moderní objektově-orientované systémy si ovšem neposílají po síti již pouhá surová data, ale typicky celé objekty. A pokud je tedy objekt sdílen, tak každá jeho změna si samo-

[8] Projekt Apache Struts je velmi známá Java technologie pro tvorbu webových aplikací. Dostupná je z této adresy: http://struts.apache.org.

[9] JSF je poměrně nová technologie pro tvorbu webových aplikací. Je specifikována pod JSR 245, viz http://jcp.org/en/jsr/detail?id=245.

[10] Projekt Apache Tomcat je nejpoužívanějším servletovým kontejnerem. Stáhnout si jej můžete z této adresy: http://tomcat.apache.org.

zřejmě žádá, aby se o ní dozvěděli i ostatní. Pro netransparentní řešení toto však znamená serializaci toho aktualizovaného objektu, neboť serializace je jediný způsob, jakým lze objekt zaslat po síti. Serializace je ovšem pomalá, neboť ona o změnách objektů nic neví. A poněvadž tedy netuší, v čem se daný objekt změnil, tak nevytváří binární podobu pouze té změny, ale vždy kompletního grafu objektu. Serializace tak pro netransparentní objektové řešení znamená přílišnou komunikaci po síti a to i tedy přesto, že většina ze zasílaných dat je zcela redundantních.

Toto se s příchodem Terracotty ovšem změnilo. Poněvadž je Terracotta transparentní a pracuje tak na úrovni JVM, může, oproti vývojáři, provádět o mnoho racionálnější rozhodnutí, co který proces v dané chvíli potřebuje. Díky transparentnosti tak Terracotta pracuje již na úrovni bajtů, respektive členů jednotlivých Java tříd a nepracuje tak již s celými objekty. Serializace tudíž u Terracotty již není potřeba a po síti se tak posílají jen opravdu ty nutné změny. Tyto změny se také již neposílají ihned každému aplikačnímu serveru (tj. klientskému JVM), ale to, co se komu v daný okamžik zašle, je řízeno tzv. „paměťovým modelem“[11].

3.2.4 Transparentnost přináší vyšší škálovatelnost

Aniž si to mnozí z nás vůbec uvědomují, tak transparentnost nám pomáhá škálovat naše Java aplikace každý den. „Garbage collector“[12] je totiž příkladem transparentní služby, která vykonává automatickou správu paměti. A právě díky transparentnosti může GC dělat svou práci lépe, než jakýkoliv sebelepší programátor, neboť jen běhové prostředí může mít opravdu ty nejpřesnější informace, jakým způsobem ho aktuálně využíváme. Může tedy dělat změny, které se vývojáři zdají příliš komplikované a objekty tak mohou být, například podle zatížení aplikace, „sbírány“ jedním algoritmem odpoledne a zcela jiným o půlnoci.

Terracotta tak zjistila, že transparentnost neřeší pouze ty složitosti, kterým by jinak programátoři čelili, ale přináší také škálovatelnost a výkon za hranicí toho, čeho je programátor vůbec svým kódem schopen dosáhnout. Jedním z příkladů běhové optimalizace výkonu je tzv. „greedy locking“[13]. Jen Terracotta může totiž za běhu rozhodnout, zda zámek má být pesimistický, centralizovaný a exkluzivní, nebo decentralizovaný a optimistický. Naopak vývojář musí vždy počítat i s tím nejhorším případem, a tak pro něj JVM bez Terracotty poskytuje, ve většině případů, pouze pesimistický exkluzivní přístup.

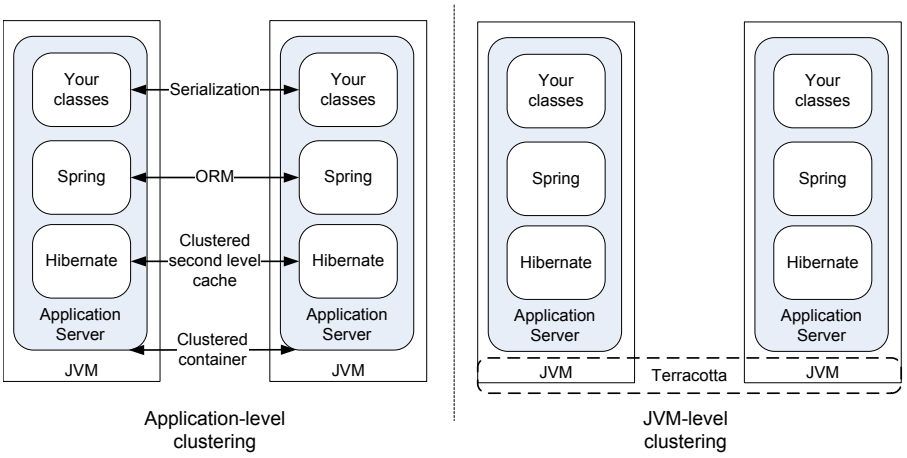
3.2.5 Terracotta, aneb transparentní klastrovací služba

Terracotta je tedy transparentní klastrovací služba, která umožňuje aplikacím rozšířit se na tolik počítačů, jak je jen potřeba. Této definici se budu podrobněji věnovat v následující

[11] Paměťový model je definován specifikací jazyka Java a více informací je o něm uvedeno například zde: http://en.wikipedia.org/wiki/Java_Memory_Model.

[12] Pokud nevíte, co to Garbage collector je, podívejte se, prosím, například zde: http://en.wikipedia.org/wiki/Garbage_collection_(computer_science).

[13] Více se o greedy locking můžete dozvědět např. zde: http://www.terracotta.org/confluence/display/devdocs/Clustered+Locks



Obrázek 5: Výhody cachování na úrovni samotného JVM [1]

kapitole, ale již teď by mělo být alespoň trochu jasné, že tam, kde jsme byli zvyklí škálovat pomocí nikdy nekončících optimalizací cachování, replikace a partitioningu a tam, kde jsme překonávali hranice mezi procesy serializací objektů a jejich následným zasíláním, nyní neděláme nic.

Ta námaha a složitost scaling-out aplikace se vytratí, pokud najdeme způsob, jakým tu meziprocesní a meziserverovou komunikaci „dostaneme“ z našeho aplikačního kódu, někde dolů, do běhového prostředí (toto ilustruje obrázek 5).

3.3 Shrnutí

Terracotta poskytuje Java aplikacím transparentní škálovatelnost s dostupností a umožňuje jim tak rozšířit se na tolik počítačů, jak je jen potřeba. Ta velmi vysoká škálovatelnost je způsoba transparentností, neboť právě ona umožňuje systému se velmi přesně rozhodnout, co mezi jednotlivými JVM a L2 zasílat. Transparentnost také nediktuje žádný programovací model a na rozdíl tak od jiných řešení majících API, může Terracotta vzít jakoukoliv již existující aplikaci a „vyškálovat“ ji ihned na libovolný počet počítačů.

V dosavadním textu jsme tedy dobře porozuměli problémům ohledně návrhu distribuovaných systémů a řekli jsme si, že je zde potřeba frameworku jakým je Terracotta. Nebudeme tedy již více otálet a v následující kapitole provedeme její podrobný rozbor. Ovšem prozatím pouze teoretický, Terracottu z praktického pohledu si necháme na kapitolu 5.

4 Terracotta - teorie

Představte si, že zavoláte na objektu metodu *wait()* a později v úplně jiném JVM vykonáte na tom stejném objektu metodu *notify()*. Představte si, že volání *notify()* překoná ty jinak nepřekonatelné hranice mezi jednotlivými JVM a to vlákno, které bylo předtím metodou *wait()* uspáno, opravdu probudí. Dále si představte, že veškeré změny, které to druhé vlákno, v tom druhém JVM, na tom daném objektu provedlo, budou navíc tomu probuzenému vláknu nyní všechny viditelné. Nakonec si představte, že zdrojový kód, který by toto umožnil, by se od toho, který byste napsali pouze pro jedno JVM, vůbec neodlišoval – tedy již žádné ústupky speciálním frameworkům, žádné volání složitých API, žádné repliky, „stubs“ či „skeletons“[14].

Jaké by to tedy bylo, kdybyste mohli automaticky sdílet haldu mezi více JVM a všechna jejich vlákna by tak mohla vzájemně spolupracovat, jako kdyby běžela pouze v jednom jediném? Představte si, čeho byste s takovými možnostmi mohli dosáhnout – například vyřešit výpočetní problém tím nejjednodušším způsobem čili programem pro jeden počítač a zároveň ho mít možnost spustit na mnoha strojích, zatímco si program myslí, že běží pouze na jednom. Nebylo by tedy opravdu skvělé mít takovéto možnosti?

Právě z takovýchto nápadů a myšlenek byla zrozena Terracotta, jež je zcela unikátní softwarové dílo, které toto všechno opravdu umožňuje.

Terracotta je „infrastrukturální“ software pro platformu Java, který nám umožňuje škálovat aplikace na tolik počítačů, jak je jen potřeba a to bez jakéhokoliv složitého kódu či databáze. Je obecně použitelným rozšířením jazyka Java, které nám nabízí nejenom extrémní škálovatelnost, ale především také jednoduchý vývoj a zároveň vysokou dostupnost. Je označována také jako tzv. „JVM-level clustering“ nebo „network-attached memory“.

Terracotta byla navržena tak, aby osvobodila vývojáře od omezení „enterprise“ vývojových frameworků jako Hibernate[15], Spring[16] a EJB[17]. At'už ale tyto frameworky aplikace používá nebo ne, ta příliš silná vazba na databázi vede vždy k nedostatku svobody v návrhu. Terracotta proto přináší zcela unikátní řešení, jak se těchto programovacích modelů a databází konečně zbavit, neboť poskytuje:

- Svobodu v návrhu – škálovatelnosti a dostupnosti již nedosahujeme striktním dodržováním nějakého programovacího modelu, ale v případě Terracotty nám k tomu stačí klasické POJO[18].
- Lineární škálovatelnost.

[14] Pokud jste někdy programovali s pomocí technologie Java RMI, jistě víte, co tyto dva pojmy znamenají. Pokud jste doposud RMI nikdy nevyužili, úvodní informace, o této zajímavé technologii, lze najít například zde: http://java.sun.com/docs/books/tutorial/rmi/index.html

[15] Hibernate je technologie realizující ORM, více informací viz http://www.hibernate.org.

[16] Spring framework je realizací IoC principu. Jeho domovská stránka je dostupná pod touto URL: http://www.springsource.org.

[17] Podniková technologie pro realizaci „business“ logiky aplikací. Je specifikována v rámci JSR 220.

[18] POJO je termín, kterým se označují všechny „obyčejné“ Java objekty, tedy ty objekty, které nerealizují žádné specifické API. Přesná definice viz například toto URL: http://en.wikipedia.org/wiki/Plain_Old_Java_Object.

- Vysokou dostupnost – na rozdíl od mnohých klastrovacích řešení, které často nabízí pouze škálovatelnost. Terracotta nejenom, že tedy dostupnost poskytuje, ale navíc k tomu, díky L2, nepotřebuje vůbec databázi.
- Stabilní operační platformu – správa Terracotty se navíc velmi podobá správě databáze, a tudíž ten, kdo umí řídit databázi, bude umět řídit i Terracottu.
- Nebývalou viditelnost napříč celým aplikačním „stackem" v rámci jediného řešení. S Terracottou tak nyní například můžeme, prostřednictvím jediného grafického nástroje (nebo JMX[19]), monitorovat aplikace nejenom na aplikační úrovni, ale také na úrovni virtuálního stroje a operačního systému.

Terracotta tedy pro aplikace představuje „neviditelný" systém sestávající z dvouúrovňového cachování L1/L2, kde L1 je transparentně instalováno do každého aplikačního JVM (jakkým způsobem je tohoto docíleno, popisuje následující kapitola) a L2 běží na dedikovaném serveru[20]. L2 tak uchovává sdílené objekty, jejichž stav je, pro rychlejší přístup, pak cachován dle potřeby v dílčích L1. Kdykoliv tedy já, jako aplikace, modifikuji stav nějakého sdíleného objektu, pak tyto úpravy (pro mne zcela transparentně) nejprve putují do L1 a až poté, v nějaký příhodný okamžik, jsou odtud zaslány do L2. L2 je také koordinátorem veškerých aplikačních aktivit, neboť kdykoliv si aplikace přeje pracovat s nějakým sdíleným objektem, L2 mu vždy nejprve zašle jeho aktuální stav (opět pro aplikaci zcela transparentně).

Ta jedinečnost Terracotty tudíž spočívá právě v té transparentnosti celého L1/L2 systému. Jelikož zde tedy není vůbec žádné API a tento systém je tak pro mne, jako aplikačního programátora, zcela neviditelný, mohu zacházet se sdílenými, respektive vzdálenými objekty jako s lokálními. Programování pro Terracottu tak probíhá pouze deklarativním způsobem, kde pouze nastavím, jaké aplikační objekty si přeji sdílet. Kdykoliv tedy spustím novou instanci mé aplikace, tak pokaždé, když se ona pokusí vytvořit novou instanci sdíleného objektu, tak Terracotta, respektive příslušná L1, se nejprve podívá, zda v L2 již stav tohoto objektu náhodou neexistuje. Pokud ano, tak tento stav bude z L2 nejprve načten a následně zkopírován do mé lokální, právě vytvářené instance. Z tohoto důvodu tak v Terracottě neexistují repliky, neboť všechny aplikační JVM pracují vždy současně pouze s tím jediným globálním stavem objektu. Z technologického hlediska tak zde není potřeba ani serializace, neboť Terracotta může pracovat již na úrovni bajtů. Mezi L1 a L2 se tak přenášejí pouze vždy změny a nikdy ne stav celého objektu.

Data, respektive objekty jsou sdíleny proto, aby k nim mohlo být, ať už z jakéhokoliv důvodu, přistupováno souběžně. Ať už tedy jednotlivé aplikační JVM sestávají z více vláken nebo pouze jednoho, ke sdíleným objektům je i tak přistupováno vždy současně. Tento paralelní přístup musí však být samozřejmě nějak synchronizován, ale jak já, jako aplikace, mohu Terracottě říci, že si přeji exkluzivní přístup k nějakému sdílenému objektu? Bez API to samozřejmě nejde, ovšem i zde se Terracottě, překvapivě, nic oznamovat

[19] JMX je technologií pro monitorování a správu Java aplikací. Je specifikována v rámci JSR 3.

[20] V produkčním prostředí je L2, pro zaručení dostupnosti a škálovatelnosti, typicky spravována mnoha servery, které vytvářejí tzv. Terracotta Server Array

nemusí. Mně totiž stačí, jako každému jinému vícevláknovému programu, přistupovat ke sdíleným objektům pouze pomocí synchronizačních technik, které nám Java již nabízí. Terracotta totiž přináší zcela unikátní a dosud nevídané řešení, které zajišťuje platnost klíčových slov *volatile* a *synchronized*, respektive veškeré sémantiky paměťového modelu (popisováno později), i v rámci celého klastru. Kde ostatní vývojáři tak, pro dosažení škálovatelnosti, musí počítat se složitostmi počítačové sítě, pro uživatele Terracotty znamená i ten sebetěžší distribuovaný systém vždy pouze program pro jeden počítač. Každý distribuovaný systém běžící nad Terracottou je tak implementován jako pouhá vícevláknová a jednoprocesní aplikace, která škálovatelnosti dosahuje paralelním přístupem mnoha jejími instancemi, či instancemi její částí ke svým sdíleným objektům.

Ve výsledku tak aplikace již nemusí řešit tu složitou meziprocesní komunikaci a veškerý paralelismus tak pro ně spočívá pouze ve vláknech a jejich správné synchronizaci. Distribuovaný systém tak již například vytvoříme pouhou jednovláknovou aplikací, kterou akorát spustíme na více serverech najednou.

Možná již tušíte, že za veškerou transparentností stojí automatická modifikace bytecode aplikačních tříd, jejichž objekty je potřeba sdílet. Je tomu opravdu tak a Terracotta skutečně zachází s instrukcemi JVM jako *getfield* a *putfield* či *monitorenter* a *monitorexit*[21]. O modifikaci bytecode pojednává ale až následující kapitola, a proto se nyní budeme zabývat rozborem definice vlastního frameworku.

4.1 Definice frameworku

Jak jsem se již zmínil, Terracotta je transparentní klastrovací služba pro Java aplikace. Její JVM-level clustering poskytuje aplikacím jednoduchý, škálovatelný a vysoce dostupný „svět“, ve kterém mohou běžet.

Abychom lépe porozuměli této definici, musíme si nejprve definovat, co to vůbec „transparentní klastrovací služba“ je a vysvětlit si pojmy jako „jednoduchost“, „škálovatelnost“ a „vysoká dostupnost“:

- Transparentnost – vzpomeňte si na úvodní kapitolu, ve které jsem říkal, že jeden z hlavních cílů každého distribuovaného systému je být svým uživatelům transparentní. V případě Terracotty je uživatelem vývojář, kterému Terracotta ne/zajišťuje tyto typy transparentnosti[22]:
  - Transparentnost přístupu – skrývá rozdíly v reprezentaci dat na různých platformách a také to, jakým způsobem je ke zdroji přistupováno
    - Pro Terracottu jsou zdroje javovské objekty, jejichž reprezentace a programátorský přístup je zcela transparentní. Poněvadž o existenci L1 a L2 nemá programátor ani ponětí, nemůže jednoduše vědět, jakým způsobem jsou v nich objekty reprezentovány. Transparentnost přístupu je také zřejmá,

[21] Více si o těchto instrukcích můžete přečíst zde: http://java.sun.com/docs/books/jvms/second_edition/html/Instructions.doc.html

[22] Typy transparentností převzaty z [2].

neboť díky tomu, že zde není žádné API, jediným rozhraním je Java samotná. Se sdílenými objekty tak programátor zachází úplně stejně jako s lokálními a to pouze prostřednictvím jazykových konstrukcí (operátor *new*, oddělovač „tečka“, klíčová slova *volatile* a *synchronized*, metody *wait* a *notify* apod.).

- Transparentnost lokace – skrývá umístění zdroje
  - Opět, není zde API, ale jenom Java samotná. Programátor (myšleno v samotném Java zdrojovém kódu, kde je práce se všemi typy objektů naprosto stejná) tudíž neví, zda pracuje s objektem lokálním anebo sdíleným a ví pouze to, že objekt je umístěn někde v paměti. Absolutně tedy netuší, že objekt může být sdílen a jeho stav tak navíc uložen v L1, respektive L2.
- Transparentnost migrace – skrývá skutečnost, že zdroj může být přemístěn do jiné lokace
  - Znovu, není zde API a objekt tak, bez vědomí vývojáře, může být přesouván kdekoliv. Ať už tedy garbage collectorem v lokální paměti nebo eventuálně na úrovni L2. Nejen totiž samotná aplikace, ale také L2 se typicky skládá z mnoha počítačů, které replikací sdílených objektů zajišťují jejich vysokou dostupnost. Platí, že Terracotta „udělá“ aplikaci vysoce dostupnou a škálovatelnou, pokud je ona sama operátorem nakonfigurována vysoce dostupnou (stejně jako v případě databází).
- Transparentnost relokace - skrývá skutečnost, že zdroj může být přemístěn do jiné lokace, i když je aktuálně používán
  - Naposled, chybí zde API, a proto se sdílené objekty nijak neliší od těch lokálních. A jelikož je tedy sdílený objekt obyčejným lokálním objektem, je i na něm vykonávána automatická správa paměti. Poněvadž ale garbage collector přemisťuje, za běhu aplikace, objekty v paměti, je z tohoto důvodu transparentnost relokace zajištěna již samotným JVM.
- Transparentnost replikace – skrývá skutečnost, že zdroj je replikován
  - Již jsem říkal, že se L2 obvykle sestává z pole serverů, kde sdílené objekty jsou, pro zajištění dostupnosti, replikovány. L2 je transparentní a tudíž i replikační transparentnost je zajištěna.
- Transparentnost selhání – skrývá skutečnost, že nastala chyba a byla provedena obnova zdroje
  - Terracotta tuto transparentnost nabízí a to díky replikaci. Poněvadž jsou sdílené objekty kopírovány, může Terracotta zabránit pozastavení aplikace i v případě výpadku nějakého L2 serveru. Replikace také značně snižuje riziko ztráty sdíleného objektu, neboť v případě poškození určitého perzistentního média, je objekt uložen naštěstí ještě na dalších médiích.
- Transparentnost souběžného přístupu – skrývá skutečnost, že zdroj může být současně sdílen několika soupeřícími uživateli

  - I přesto, že se sdílený objekt nijak neliší od toho lokálního, tak každý přístup k němu musí být, programátorem, explicitně synchronizován. Synchronizaci, klíčovým slovem *synchronized* anebo *volatile*, musí programátor provádět proto, aby Terracotta přesně věděla, na jakých místech musí provést instrumentaci[23], a zajistit tak platnost synchronizace i v rámci klastru. Transparentnost souběžného přístupu je tedy jediným typem transparentnosti, který Terracotta nenabízí, neboť programátor musí vždy vědět, že pracuje se sdíleným objektem, a veškerou práci s ním (tzn. i čtení) tak provádět v synchronizovaném bloku.

  Transparentnost tak pro vývojáře znamená, že aplikace, navržená pro běh nad Terracottou, bude fungovat i v případě, že Terracotta nebude vůbec dostupná. Toto ovšem neznamená, že je Terracotta vhodná úplně na všechny typy aplikací, nebo že je Terracotta „absolutně" neviditelná a nevyžaduje vůbec žádné změny do aplikačního kódu. Transparentnost pro uživatele Terracotty totiž znamená svobodu a čistotu v návrhu, což je velmi výhodné, neboť aplikace může být taková, jak si přeje jenom on sám.

- Klastrování – již v úvodní kapitole jsem nazval distribuovaný systém klastrem. I to ukazuje, že klastrování má mnoho definic, ale obecně se dá říci, že značí vzájemnou komunikaci mnoha počítačů za dosažením určitého cíle. Terracotta se nazývá klastrovací službou a vzhledem k „pouhému" klastrování (na úrovni zdrojového kódu) je unikátní v tom, že nám umožňuje škálovat i takovou aplikaci, která nemá v sobě jedinou klastrovací logiku. Ke klastrování tak již nedochází na úrovni aplikace, ale pod ní, v prostředí JVM. Toto přenáší to břemeno klastrování z konceptu architektury na službu, na kterou se aplikace mohou, v produkčním prostředí, spolehnout.

- Jednoduchost – a složitost značí, jaká opatření musí programátor udělat na své cestě ke škálujícímu systému. Typickým příkladem (složitosti) je rozhraní *Serializable*[24], které často musí třídy implementovat, aby jejich objekty mohly být posílány po síti. My už ale víme, že u Terracotty toto není potřeba, a že každý distribuovaný systém je tak pouze otázkou návrhu jednoprocesního programu.

- Škálovatelnost – škálovatelnost můžeme měřit podle tří dimenzí[25], ale často ji, ačkoliv trochu nepřesně, spojujeme pouze s výkonnostními problémy (viz [2]). Škálovatelnost je tím nejdůležitějším cílem každého distribuovaného systému a troufám si tvrdit, že bez ní nemůže být žádný systém úspěšný. Je výsledkem malého zpoždění

[23] Instrumentace označuje proces modifikace stávajícího bytecode za účelem např. monitorování aplikačního výkonu.

[24] Pokud libovolná Java třída realizuje rozhraní *Serializable*, značí tím, že je možno ji bez problému transformovat na pole bajtů a obráceně.

[25] První dimenze je velikost, což znamená, že do systému můžeme snadno přidávat nové zdroje. Druhá je geografická dimenze, která značí, že jednotlivé prvky systému se mohou nacházet daleko od sebe, a konečně třetí je administrativa. Administrativní škála znamená, že systém je stále snadno spravovatelný, a to i tehdy, pokud je již rozšířen přes mnoho organizací.

a vysoké propustnosti. Aplikace může totiž velice rychle odpovídat na každý dotaz uživatele, ale zároveň být schopna, v danou chvíli, zpracovávat pouze jediný (malé zpoždění a nízká propustnost). Naopak, což je typický případ databází, aplikace může být pomalá v odpovědích, ale pro změnu zvládat tisíce požadavků najednou (vysoké zpoždění a vysoká propustnost). Na rozdíl od klasických databází, Terracotta pomáhá optimalizovat jak zpoždění, tak propustnost.

- Dostupnost – dostupnost již také známe a tak víme, že znamená, že i ta nejmenší část sdílených dat musí být zapsána na pevný disk. Je důležitá, neboť chrání data v případě výpadku proudu nebo chyby aplikačního procesu.

Poté, co již známe všechny potřebné pojmy, se nyní můžeme blíže podívat na slovní spojení „transparentní klastrování “. K jejímu pochopení nám pomohou analogie k Terracottě, které již v prostředí datových center existují.

4.1.1 Terracotta a její podoba s NAS

Terracotta jako transparentní klastrovací služba může být použita k mnoha účelům jako například k replikaci relací, distribuovanému cachování, grid computingu[26] apod. Transparentnost, tak jak jsme si ji definovali, dovoluje aplikacím záviset na Terracottě, aniž by se na ni explicitně odkazovaly. Existují ale i další transparentní služby, na které se aplikace spoléhají, aniž by je musely používat ve svém zdrojovém kódu.

Zřejmě nejanalogičtější transparentní službou k Terracottě je úložiště souborů. Aplikace totiž pracuje se soubory, aniž by věděla, kde se aktuálně nacházejí. Ať už jsou tedy uloženy lokálně, anebo vzdáleně, na pevném disku či CD, program s nimi pracuje vždy konzistentním způsobem. Soubory tak představují populární mechanizmus pro ukládání aplikačních dat, neboť bez jakékoliv změny programového kódu může operátor libovolně měnit jejich lokaci nebo nastavit úplně nový systém souborů.

Terracotta poskytuje stejné možnosti „Javovské“ haldě. I přesto, že ale soubory nejsou zcela bezchybným příkladem, i tak nám lépe pomohou pochopit výhody, které s sebou transparentní služby přináší.

Souborové API můžeme omezit na tyto čtyři funkce: *open()*, *seek()*, *read()* a *write()*. Ačkoliv tak díky nim není práce se soubory zcela transparentní, to, co se však skrývá pod nimi, je starostí už pouze operátora. On je tak může implementovat různými způsoby a dle povahy aplikace vybrat mezi systémy jako například HPFS[27], ZFS[28] či NTFS[29].

Pokud se tedy bude program držet jen tohoto standardního API, nebude muset být nikdy přepsán a to ani v případě, kdy se operační systém rozhodně kompletně aktualizovat svoji správu souborů.

Terracotta poskytuje také abstrakci, ne však již pro soubory, ale pro objekty. Její uživatelé si tak např. nepotřebují domýšlet, jakým způsobem se objekty pohybují po síti,

[26] Více o gridech v podkapitole 6.1.

[27] Souborový systém vytvořený jako náhrada systému FAT v OS/2.

[28] Poměrně nový souborový systém vytvořený společností Sun Microsystems, Inc.. Jeho devízou je především podpora velkých kapacit a různé vlastnosti zvyšující spolehlivost.

[29] Souborový systém využívaný v operačních systémech založených na Windows NT.

ani nemusí psát kód pro rozdělení dat mezi více procesů. Stejně jako operační systém může kdykoliv za běhu změnit správu souborů, tak i Terracotta může provádět různé optimalizace, aniž by se musela aplikace jakkoliv změnit.

4.1.1.1 Paralely Terracotty a síťového úložiště

Síťové úložiště neboli NAS[30] je zařízení, do něhož jsou ukládány soubory. Soubory tak již nemusí být pro aplikaci lokální, ale mohou být tímto způsobem uloženy vzdáleně. Operátor tak již nemusí draze spravovat všechny aplikace najednou, ale může se soustředit pouze na jeden kus hardware (myšleno datové úložiště). Jednoduše tak zajistí vysokou dostupnost, kdy obsah úložiště pouze zálohuje pomocí známých technik jako replikace dat na dedikovaný server nebo využití páskových robotů. Síťové úložiště také dobře škáluje a to i tehdy, aniž bychom koupili lepší CPU, anebo dražší serverový hardware.

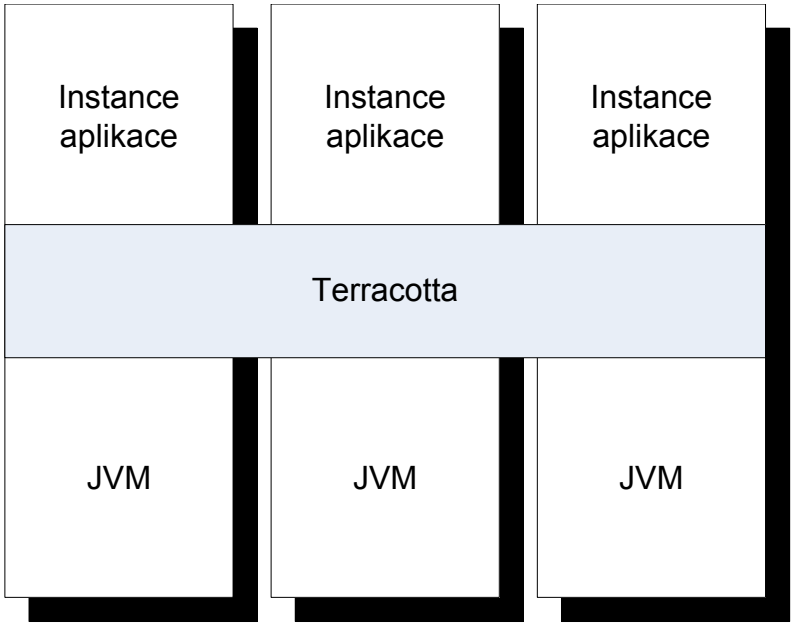
Terracotta poskytuje operátorům podobné výhody jako síťové úložiště. Objekty jsou totiž v Terracottě uloženy vzdáleně a pro rychlejší přístup zároveň cachovány uvnitř každého klientského JVM. Operátor se tak, stejně jako v případě souborového systému, může soustředit pouze na optimalizaci Terracotta serveru (který, jak již víme, je realizací L2) a zaručit tak jeho vysokou dostupnost. Tímto způsobem může aplikační programátor používat objekty stejně jako souborové API a nechat jejich efektivní a bezpečnou správu pouze na Terracottě.

Poslední paralelou mezi Terracottou a soubory je jejich totožný způsob klastrování, tedy způsob komunikace a synchronizace. Jak Terracotta, tak soubory klastrování realizují pomocí exkluzivních a sdílených zámků a distribuovaná aplikace založená na souborech, by jednoduše bez „zamykání“ nemohla fungovat. Bez zámků by totiž počítače systému jednoduše nevěděly, kdo z nich se souborem aktuálně pracuje. Zámky jsou proto prostředkem pro synchronizaci a pomocí nich se jednotlivé počítače mohou snadno domluvit. Bez sdíleného úložiště a dalších technologií[31] by však tento jednoduchý způsob nefungoval. Bylo by totiž najednou úkolem programátora rozdělovat data mezi procesy, k čemuž by ale docházelo (např. pomocí „soketů“) vysoko nad úrovní souborového systému. Operátoři by tak ve výsledku jednoduše ztratili možnost spravovat tyto distribuované aplikace pouhou správou jejich úložiště.

Soubory jsou tak dobrým příkladem transparentní klastrovací služby. Vývojáři totiž zapisují do souborů, kdykoliv a jak chtějí a stejně tak operátoři tyto soubory ukládají. Ve stručnosti se dá říci, že transparentní služby jako NFS, respektive síťové úložiště pomáhají vývojářům realizovat aplikace takovým způsobem, jakým si přejí. Operátorům zase umožňují vytvořit opravdu škálovatelné a vysoce dostupné produkční prostředí pro sdílená data.

[30] NAS neboli Network-attached storage je datové úložiště připojené k místní síti LAN. Jeho výhodou oproti ukládání dat na běžném PC nebo aplikačním serveru je jednoduchost konfigurace a možnost sdílení dat v síti.

[31] Například technologií jako NFS. Více informací viz RFC 5661.



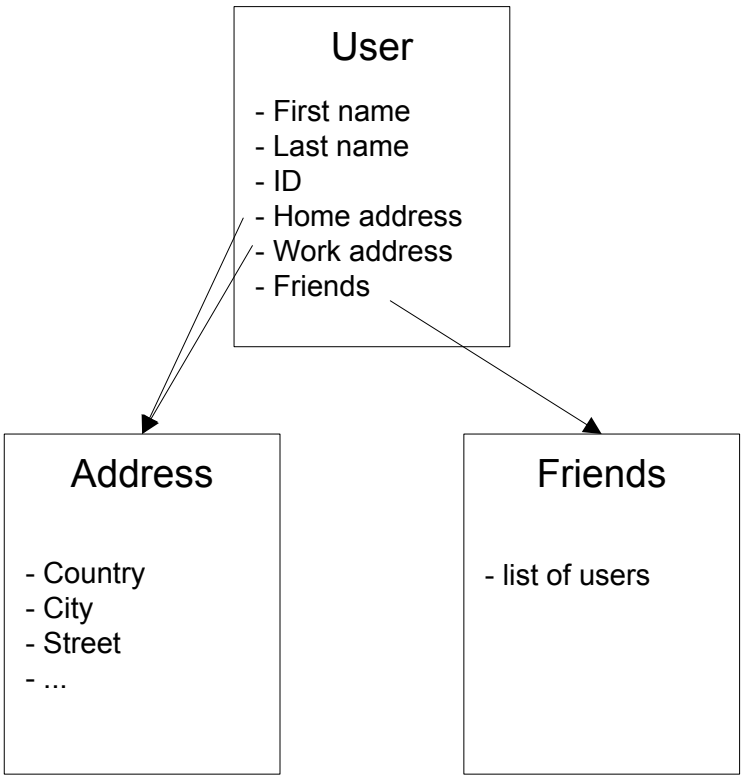
Obrázek 6: Terracotta pracuje mezi aplikací a JVM [1]

4.1.2 Transparentnost umístění objektů v paměti

Na rozdíl od souborů nemá Terracotta vůbec žádné API. Pro aplikace je tak virtuálním strojem a pro virtuální stroj aplikací. Toto také pěkně ilustruje obrázek 6.

Na rozdíl však od souborového API, kde tým operátorů může přiřadit určitému adresáři síťový souborový systém a tím sdílet vše, co je v něm uloženo, tak Terracotta pracuje s objekty, což určuje i jejich transparentnost umístění v paměti. Ona tak sama musí rozhodnout, co sdílet a co ne a na rozdíl od jmen souborů, je pro ni jediným rozhraním samotná objektová orientace jazyka Java. Paralela s „mountováním“ vzdálených repositářů však platí i zde, neboť objekty vytvářejí struktury (pomocí svých proměnných, které odkazují na další objekty, které zase odkazují na další objekty atd.) stejně jako adresáře a soubory. Vývojáři tak vytvářejí modely (grafy) objektů, které jsou pak klíčem pro komunikaci s Terracottou a obsahují vše, co Terracotta i virtuální stroj potřebují znát, pro jejich správné „namapování “ do paměti.

Jako příklad uvedu třídu *User* (viz obrázek 7), kterou sice můžeme modelovat jako jednu velkou, všeobsahující třídu, avšak většinou to neděláme a její model sestavujeme z více tříd (stejně jako na obrázku). Je zřejmé, že když vývojář pracuje se sdíleným *User* objektem, tak automaticky očekává, že objekty, na které odkazují jeho proměnné *Home address*, *Work address* a *Friends* jsou také sdíleny. V opačném případě by aplikace sice měla ke sdílenému objektu *User* přístup, ale byl by jí na nic, neboť by již neměla k dispozici jeho data. Terracottě tak proto stačí pouze říct, že si přejeme sdílet objekt *User* a ona si již sama, z jeho definice, odvodí vše potřebné. Toto je analogické k síťovým souborovým systémům, neboť stejně jako oni, i Terracotta „namountuje“ objekt ze vzdáleného úložiště do lokální



Obrázek 7: Třída *User* a její vazby [1]

paměti. S objektem tak bude uživatel pracovat lokálně, ovšem provedené změny budou ukládány vzdáleně. Jakým způsobem se v Terracottě grafy objektů vytvářejí, je uvedeno v následující kapitole.

4.1.3 Transparentnost a klastrování dohromady

Terracotta, jako transparentní klastrovací služba, pomáhá vývojářům sdílet část jejich lokální paměti. Každý objekt jednoho JVM tak může být viditelný i ostatním a to bez ohledu na to, zda tyto další Java procesy běží na stejném nebo úplně jiném stroji.

Aby však mohli vývojáři těžit z výhod, které jim Terracotta nabízí, musí mít každý sdílený objekt, ve všech virtuálních strojích, stejné *ID*. Stejně jako v případě NFS, kde, pokud aplikace chtějí pracovat se stejným souborem, musí obě „namountovat“ totožný souborový systém a používat také identický název souboru. Každý sdílený objekt tak v Terracottě obdrží něco jako je název souboru, který pak slouží pro jeho identifikaci v rámci klastru (tj. Terracotta serveru a jednotlivých klientských JVM, na kterých běží instance aplikace - popřípadě její části. Aplikace, jak již víme, se totiž spouští opakovaně pro dosažení lepšího výkonu. Toto je ale obvyklé nejen u Terracotty, nýbrž i u ostatních distribuovaných systémů).

K tomu, aby však daný JVM viděl instanci, kterou vytvořil jiný virtuální stroj, musí Terracotta překonat více překážek. Zaprvé, objekty vytvořené v jednom adresním prostoru, mají oproti jinému procesu různé adresy v paměti – a tím i své identifikátory. Dále, změny provedené jednou aplikací musí být samozřejmě viditelné i ostatním.

Důležitou roli v řešení těchto problémů hraje rozsah platnosti objektu. Ne každý objekt je totiž potřeba sdílet a je také naivní se domnívat, že Terracotta sama vyhledá všechny instance třídy *User* (viz předešlá podkapitola) a začne je automaticky sdílet. Pokud tak není objekt vytvořen na vhodné úrovni, a není proto přístupný ostatním vláknům či dokonce stejnému vláknu později, není ho třeba sdílet. Typickým příkladem jsou objekty vytvořené v metodách, neboť jejich platnost je omezena pouze na dobu trvání té dané metody. Naopak objekty v kolekcích, „singletony“[32] a globální objekty jsou všechny vhodnými kandidáty pro sdílení. Obecně tak platí, že každý objekt, který je přístupný více než jednomu vláknu, může být sdílen.

Na výpisu 1 můžete zhlédnout různé platnosti objektů. V prvním případě *c1* nikdy „nepřežije“ volání metody *printAClusteredClassInstance*. Není ho tak potřeba sdílet, ačkoliv je třída *ClusteredClass* k tomu určena. Naopak *c2* je potřeba dále sledovat, neboť jeho možné sdílení závisí na další činnosti vlákna, které volá metodu *clone*. *c3* by měl být sdílen určitě, neboť je přidán do veřejné kolekce a je tak vláknům přístupný pod svým klíčem.

```
// prvni platnost: pouze v metode
void printAClusteredClassInstance( ) {
    ClusteredClass c1 = new ClusteredClass();
    c1.toString( );
}

// druha platnost: platnost i ve volajici metode
ClusteredClass clone( ClusteredClass c ) {
    ClusteredClass c2 = new ClusteredClass();
    return c2; // c2 prezije konec teto metody, tedy muze byt zajimava
}

// treti platnost: globalni, ulozeno v singletonu
static Map myMap = new HashMap();
void storeAClusteredClassInstance( ClusteredClass c3 ) {
    myMap.put( "some-id", c3 );
}
```

Výpis 1: Různé platnosti objektů [1]

Jednoznačný „cluster-wide“ název objektu a dále skutečnost, že se objekt již nemusí, potom, co byl libovolným JVM již vytvořen, znovu vytvářet, je spojeno s voláním konstruktorů. Sdílený objekt je totiž v Terracotta klastru vytvořen vždy jen jednou a volání konstruktorů na sdílených objektech je tak analogické k příkazu *mount*. Terracotta tak při volání konstruktoru sama rozhodne, zda je potřeba konstruktor opravdu zavolat, anebo v případě, že je objekt již vytvořen, nikoliv (toto se vše děje na úrovni bytecode,

[32] Třídy, ke kterým existuje nanejvýš jedna globální instance. Více informací viz návrhový vzor Singleton.

jelikož, jak jsem již ostatně také zmínil, Terracotta se do aplikací integruje právě jeho modifikací. Je tak schopna ignorovat instrukce realizující volání konstruktorů.). Například mapu *myMap* (viz výpis 1) vytvoří, v metodě *storeAClusteredClassInstance*, pouze první z klientských JVM a další ji už jen využívají. Tedy u těch se volání konstruktoru nahradí stažením sdíleného objektu z Terracotta serveru, respektive jeho dat. Více na toto téma v následující kapitole.

Další unikátní vlastností Terracotty, bez které by to ale jednoduše (transparentně) nešlo, je zamykání sdílených objektů čistě pomocí Java synchronizace. Synchronizace souvisí s paměťovým modelem Javy, který nám ve stručnosti říká (a virtuální stroj zajišťuje), že pokud určité vlákno vstoupí pod stejným zámkem do synchronizovaného bloku, tak má záruku, že uvidí i ty změny, které provedly jiná vlákna.

4.2 Být službou má své výhody

Terracotta je „transparentní klastrovací služba“. Transparentností a klastrováním jsme se zabývali doposud, a tak nyní nastal čas podívat se na poslední slovíčko „služba“.

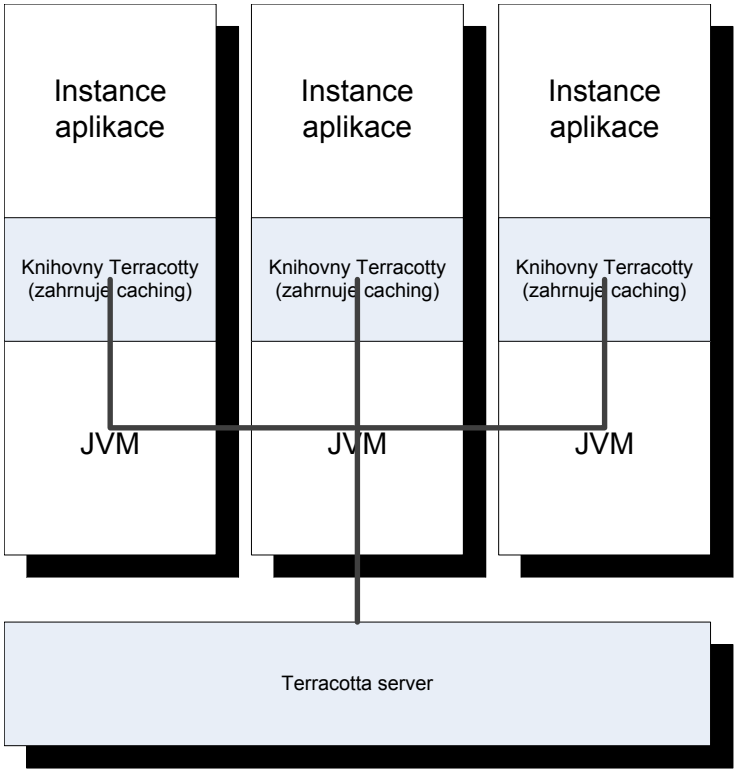
Terracotta se sestává pouze ze dvou komponent, což zobrazuje také obrázek 8. Používá architekturu klient/server, kde serverový proces představuje vlastní Terracottu a stará se tak o klastr a ostatní služby, které s ním souvisí. Na klientské straně jsou pak instalovány pouze knihovny, které (společně s cachováním a dalšími službami) zprostředkovávají komunikaci se serverem.

Díky této architektuře se může operátor soustředit pouze na server. Škálovatelnost tak znamená pouhé přidání více virtuálních strojů a vysokou dostupnost zaručíme obdobně jako u databází. Zatímco tak aplikace používají Terracottu pro dosažení vysoké škálovatelnosti a dostupnosti, tak samotná škálovatelnost a vysoká dostupnost Terracotty je úkolem operátora. Stručně řečeno, pokud operátor „udělá“ Terracottu vysoce dostupnou, Terracotta zaručí, že i aplikace, které nad ní běží, budou taktéž vysoce dostupné.

4.2.1 Dostupnost

Dostupnost nám dává záruku, že i po (chybovém či řízeném) ukončení všech Javovských procesů, budeme mít přístup ke všem sdíleným objektům i po jejich opětovném nastartování. Tuto vlastnost čili vysokou dostupnost nám Terracotta zaručuje tak, že žádná aplikace neskončí dřív, dokud operátor „neflushne“ všechny objekty uložené v paměti nebo na disku.

Proces Terracotty (tj. Terracotta server) je proto navržen tak, aby byl spolehlivý a schopný restartu. Kdyby skončil např. chybou nebo byl zastaven, bude spuštěn přesně tam, kde skončil a zatímco on znovu startuje, aplikační procesy pracují dál, aniž by zaregistrovaly nějakou nastalou chybu či výjimku. Schopnost restartu je tak výhodná především pro malé aplikace a během vývoje, ovšem větší systémy upřednostňují spíše co nejdelší bezproblémový chod. Z tohoto důvodu není klastrována pouze daná aplikace, ale i vlastní Terracotta server. Tím se pro produkční prostředí zaručí požadovaný stupeň dostupnosti a dojde i tak k rozdělení povinnosti vykonávat správu objektů daného clusteru.



Obrázek 8: Terracotta je složena pouze ze dvou komponent, Terracotta serveru a klientských knihoven. Tyto komponenty spolu komunikují přes TCP/IP [1].

4.2.2 Škálovatelnost

Terracotta je transparentní čili ke škálování dochází „za zády“ programátora. To ještě ovšem neznamená, že každá aplikace, jakkoli napsaná, bude pracovat nad Terracottou efektivně. I když programátor nepracuje s žádným API, je více způsobů, jakými lze aplikaci vylepšit a docílit tak toho, aby byla co možná nejvýkonnější.

„Vylepšení“ aplikace můžeme už dosáhnout pouhou změnou konfiguračního souboru (viz následující kapitola) a to nastavením způsobu klastrování. Další možností je změna zdrojového kódu, s jejíž pomocí můžeme občas dosáhnout poměrně významného nárůstu ve výkonu.

Celkový výkon aplikace je ale zřejmě nejvíce ovlivňován správným využíváním interních cache. To se většinou pojí s inteligentním směrováním příchozích dat a dobrým příkladem jsou tak webové aplikace. Ty totiž používají tzv. „sticky load balancing“, kde všechny požadavky náležící do jedné uživatelské relace, jsou vždy směrovány do jednoho uzlu. Tím nedochází, mezi aplikačními servery, ke zbytečnému přenosu dat a ty tak mohou být ukládány pouze do jediné cache. Výsledkem je tak velmi vysoká rychlost, neboť data nejsou aplikačním serverem získávána ze sítě, nýbrž pouze z operační paměti, respektive cache.

Caching je analogií k virtuální paměti operačního systému, neboť i zde, pokud určitá data objektu či objekt celý nejsou lokálně dostupná, musí být nejprve ze serveru získána. A je zřejmé, že pokud pro data není v cache dostatečné místo, musí se nejprve (dle určitého algoritmu, jako např. LRU či LFU[33]) nějaká méně potřebná data odebrat a případně odeslat serveru. Velikost cache je tak dalším nástrojem, kterým můžeme výrazně optimalizovat chod celé naší aplikace.

Dalšími technikami pro zvýšení škálovatelnosti jsou distribuce a replikace dat. Ty jsou především známy „databázistům“, a poněvadž je někdy obtížné se správně rozhodnout, kde a nakolik počítačů data rozdělit či replikovat, snaží se toto Terracotta provádět za nás.

4.2.3 Vyhýbání se výkonnostním překážkám

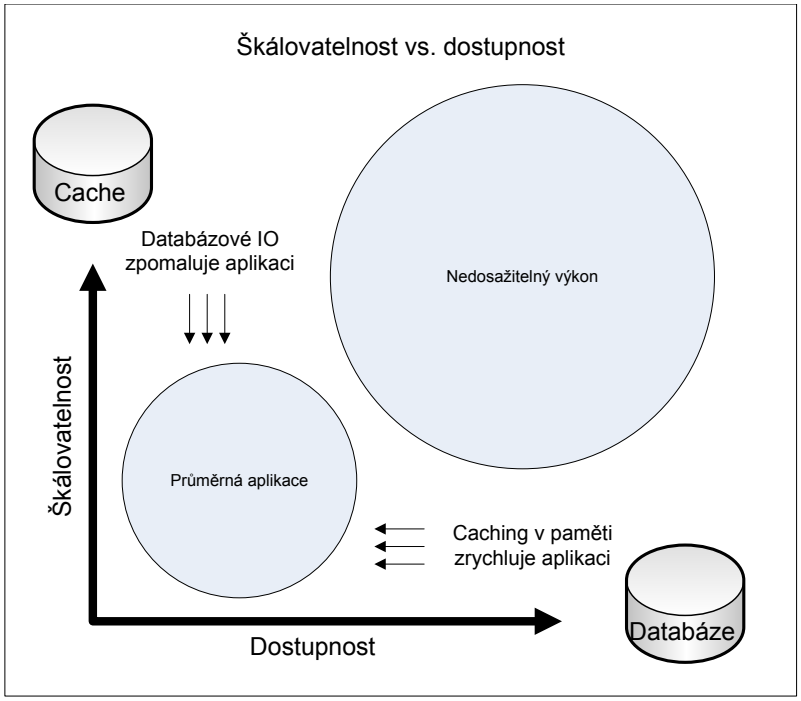
Největší problém týkající se optimalizace aplikace, spočívá v potřebě ukládat data na disk pro zaručení jejich dostupnosti a zároveň zajištění dostatečné propustnosti a co nejmenšího zpoždění. Databáze nám sice zaručí vysokou dostupnost, ale mnohdy také za cenu snížené škálovatelnosti a zpravidla vysokého zpoždění. Naopak cachováním dosáhneme vysoké propustnosti a téměř žádného zpoždění, to vše ale na úkor dostupnosti, neboť se nic nezapisuje na disk.

Tento problém ilustruje také obrázek 9, a i když lze všech požadovaných vlastností dosáhnout pouze pomocí databáze, u velkých (celosvětových) distribuovaných aplikací to téměř vždy znamená použití drahých a složitých nástrojů jako např. Oracle Real Application Clusters[34]. Terracotta ovšem nabízí o mnoho jednodušší a také levnější řešení, které je navíc často i mnohonásobně rychlejší než samotná databáze. Následující výčet technik naznačuje, jakým způsobem toho Terracotta dosahuje:

- Terracotta se vždy snaží data načíst nejprve z cache a až tehdy, pokud nejsou dostupná, je načítá ze serveru (tj. z Terracotta serveru, tedy z L2 cache).
- Terracotta nenačítá z disku nikdy víc, než potřebuje a čte z něj téměř v konstantním čase.
  - jednotlivé datové členy objektů ukládá na disku blízko u sebe a pokud možno, tak nejlépe do jednoho bloku.
- Terracotta zapisuje pouze to, co bylo změněno a změny zapisuje v dávkách.
- Terracotta zapisuje stylem „append-only“, čili objekty nejsou na disku nějak tříděny, indexovány či přemisťovány/nefragmentovány.

[33] Tyto algoritmy se můžete naučit například z této URL: http://en.wikipedia.org/wiki/Cache_algorithms.

[34] Domovskou stránku této technologie můžete nalézt zde: http://www.oracle.com/global/cz/database/rac_home.html.



Obrázek 9: Škálovatelnost vs. dostupnost [1]

4.3 Případy užití

Terracotta je unikátní technologie, která je pro aplikaci téměř transparentní a může být tak využita mnoha způsoby. Její cílovou platformou jsou především podnikové aplikace, kde s výhodou využijeme např. tyto případy užití:

- Distribuovaná cache objektů - jelikož je Terracotta schopna sdílet téměř jakékoliv objekty, může si každá aplikace, pomocí Java Collections API, velice snadno vytvořit svou vlastní distribuovanou cache. Terracotta také umí sdílet kolekce z balíčku *java.util.concurrent*, a tak se navíc nemusíme vůbec starat o správnou synchronizaci.
- Zbavení se databáze - s Terracottou se, v naší distribuované aplikaci, bez databáze zcela obejdeme. Data totiž můžeme uchovávat např. ve sdílených kolekcích a namísto SQL využít standardních Java prostředků. Terracotta nám potom sama zajistí jejich vysokou dostupnost, neboť data, v tomto případě objekty, automaticky ukládá na disk.
- Řízení zátěže - K náročným výpočtům a zpracování dat již nemusíme s Terracottou využít nějakého složitého výpočetního gridu. Vystačíme si totiž pouze s vlákny a sdílenou Java kolekcí (např. *java.util.Queue*) a poté program pouze spustíme na více uzlech najednou.

5 Terracotta - prakticky

Dost již bylo teorie, ukažme si Terracottu konečně i z praktického pohledu. V této kapitole vám tedy ukážu, jakým způsobem Terracotta vlastně funguje a co vše musí programátoři udělat, aby nad ní vytvořili fungující distribuovaný systém.

Již víme, že distribuovaný systém se pomocí Terracotty vytvoří pouhým sdílením objektů. Nicméně, prozatím jsem tuto unikátní vlastnost Terracotty neilustroval na žádném příkladu, a tak v první části této kapitoly si z jednoduché aplikace, napsané pouze pro jedno JVM, vytvoříme její distribuovanou podobu.

Neboť ke klastrování dochází u Terracotty na úrovni JVM, nekomunikujeme s ní prostřednictvím API, ale pomocí konfiguračního souboru. Ten se nazývá *tc-config.xml* a v něm tedy Terracottě řekneme, jaké objekty si přejeme sdílet, u jakých tříd bude vykonána instrumentace apod. Druhá část je tak věnována tomuto konfiguračnímu souboru a popíši tedy na něm vše, co je potřeba znát k tvorbě distribuovaných systému nad Terracottou.

5.1 Aplikace HelloClusteredWorld

Jak jsem řekl, v této podkapitole je na jednoduchém příkladu ilustrováno, že s Terracottou nám, k dosažení distribuovaného systému, stačí sdílet objekty.

Zdrojový kód příkladu je uveden na výpisu 2 a, jak můžete vidět, je opravdu velice jednoduchý. Program totiž pouze v nekonečné smyčce plní statický buffer řetězcem „*Hello Clustered World!*“. V každé iteraci uloží do pole vždy jedno písmeno, poté buffer zobrazí a nakonec se uspí na 100 milisekund. To, který znak se má do bufferu aktuálně uložit, určuje proměnná *loopCounter*, respektive výraz *loopCounter*++ % *length*. Program opravdu neskýtá žádnou záludnost a po jeho spuštění tak skutečně uvidíme nejprve písmeno „*H*“, následně znaky „*He*“ a každou 22. iteraci pak kompletní zprávu „*Hello Clustered World!*“. Toto lze vidět i na výpisu 3.

```
public class HelloClusteredWorld {
    private static final String message = "Hello Clustered World!";
    private static final int length = message.length();
    private static char[] buffer = new char [length];
    private static int loopCounter;

    public static void main( String args[] ) throws Exception {
        while( true ) {    synchronized( buffer ) {
                int messageIndex = loopCounter++ % length;
                if ( messageIndex == 0 ) java.util.Arrays. fill (buffer, '\u0000');
                buffer[messageIndex] = message.charAt(messageIndex);
                System.out.println( buffer );
                Thread.sleep( 100 );
        } }
    }
}
```

Výpis 2: Třída *HelloClusteredWorld* [1]

```
H
He
Hel
Hell
Hello
Hello
Hello C
Hello Cl
Hello Clu
Hello Clus
Hello Clust
Hello Cluste
Hello Cluster
Hello Clustere
Hello Clustered
Hello Clustered
Hello Clustered W
Hello Clustered Wo
Hello Clustered Wor
Hello Clustered Worl
Hello Clustered World
Hello Clustered World!
H
He
Hel
Hell
Hello
 ...
```

Výpis 3: Výstup třídy *HelloClusteredWorld*

Již víme, že Terracotta pomáhá aplikacím rozšířit se mezi více procesů. Jaké chování bychom ale měli očekávat od distribuované verze našeho modelového příkladu? Samozřejmě chceme docílit toho, aby bylo možno spustit aplikaci vícekrát, přičemž všechny spuštěné instance by plnily pouze jeden, globálně přístupný, buffer. Tím, že by byl plněn pouze jeden buffer, dosáhli bychom distribuovaného systému.

Jestliže bychom spustili například tři instance, k naplnění celého bufferu by nyní stačilo, aby každá instance, z celkového počtu znaků, uložila do bufferu pouze jednu třetinu (samozřejmě za předpokladu, že by procesy byly plánovány naprosto férově), v případě čtyř instancí už jen jednu čtvrtinu, pěti jednu pětinu, osmi jednu osminu atd.

Tohoto požadovaného chování s Terracottou docílíme velmi jednoduše a to navíc bez jakéhokoliv zásahu do zdrojového kódu. Vystačíme si totiž pouze s několika málo řádky konfiguračního souboru, ve kterém Terracottě sdělíme vše, aby mohla *HelloClusteredWorld* distribuovat.

V našem případě bude konfigurační soubor obsahovat údaje, které jsou zobrazeny na výpisu 4. Konfiguračními elementy se podrobněji zabývám až v následující části, nyní vám jen stačí vědět, že prvními dvěma řádky specifikuji, aby proměnné *buffer* a *loopCounter* byly sdíleny. Sdílením tak budou viditelné všem instancím aplikace a jejich hodnoty

budou proto ukládány do L2. Kdykoliv tedy v *main()* přistupujeme k těmto sdíleným objektům, je nutné L2 kontaktovat. Poněvadž ale tuto komunikaci neprovádíme my, nýbrž Terracotta, musí se modifikovat bytecode třídy *HelloClusteredWorld*. Toto provedeme elementem *instrumented-classes*, který je taktéž ve výpisu uveden. Tímto budou do bytecode třídy umístěny instrukce, které budou, při manipulaci se sdílenými proměnnými, komunikovat s L2.

```
<field−name>HelloClusteredWorld.buffer</field−name>
<field−name>HelloClusteredWorld.loopCounter</field−name>

...

<instrumented−classes>
   <include>
      <class−expression>HelloClusteredWorld</class−expression>
   </include>
</instrumented−classes>

...

<locks>
   <autolock>
      <method−expression>void HelloClusteredWorld.main(java.lang.String[])</method−
          expression>
   </autolock>
</locks>

...
```

Výpis 4: Fragmenty konfiguračního souboru aplikace *HelloClusteredWorld* [1]

Poslední element souvisí se synchronizací a vysvětlím jej až později. Ve výpisu 4 je uveden pouze proto, jelikož by bez něj distribuovaná verze aplikace *HelloClusteredWorld* nefungovala.

Jak můžete vidět, krom proměnné *buffer*, je sdílena i *loopCounter*. Její sdílení je stejně tak důležité, neboť programu určuje, které písmeno aktuálně uložit. Pokud by však sdílena nebyla a každý proces ji tak inkrementoval nezávisle na ostatních, virtuální stroje by jednoduše ukládaly znaky na neplatné pozice a na konzolích bychom viděli pouze znetvořené řetězce.

Konfigurační soubor již máme, můžeme tedy *HelloClusteredWorld* nad Terracottou spustit. Pokud si *HelloClusteredWorld* napoprvé spustíte, na výstupu (oproti výpisu 3) byste ještě neměli zaregistrovat žádnou změnu (jak skutečně *HelloClusteredWorld* spustit, je popsáno až později). Opět tedy uvidíte, jak se na každém řádku postupně přidává jedno písmeno za druhým. Změna ovšem nastane, pokud aplikaci spustíte i podruhé. Takto se totiž najednou na plnění bufferu začnou podílet již hned dva procesy. Na výstupu každého virtuálního stroje tak již neuvidíme „*H*“ a potom „*He*“, ale každý řádek se nyní bude lišit již o dvě písmena. Dva procesy tak znaky ukládají dvojnásobnou rychlostí a každé další

spuštění nám tuto rychlost zvýší lineárně. Výstup druhého JVM si můžete prohlédnout na výpisu 5.

```
Hel
Hello
Hello C
Hello Clu
Hello Clust
 ...
```

Výpis 5: Výstup třídy *HelloClusteredWorld* (na jednom JVM) při dvou spuštěných JVM

5.2 Konfigurace Terracotta aplikací

Připomeňme si architekturu distribuovaného systému nad Terracottou (viz obrázek 8). Každý systém se skládá z minimálně jednoho Terracotta serveru (L2) a jedné či více instancí aplikace (L1). Každá instance se připojuje právě k jednomu serveru a jak aplikace, tak server jsou konfigurovány pouze jedním konfiguračním souborem. Tento soubor se zpravidla nazývá *tc-config.xml*. Více se o tom ještě zmíním později, ale start serveru i aplikací se provádí pomocí dávek, které jsou součástí distribuce Terracotty. Právě těmto skriptům předáváme jako vstupní parametr soubor *tc-config.xml*, který tedy obsahuje jak serverové, tak aplikační konfigurační údaje. Server i aplikace si z něj načtou vždy jen to, co potřebují.

tc-config.xml je XML soubor a definuje ho XML schéma[35]. V následujících podkapitolách se vám tedy pokusím vysvětlit princip jeho hlavních elementů, jež vám pro tvorbu většiny Terracotta aplikací budou dostačovat. Tři z nich popíši podrobněji (*roots*, *instrumented-classes* a *locks*), neboť souvisí se třemi hlavními koncepty, na kterých je Terracotta vlastně postavena.

V následujícím textu nepopíšu úplně všechny elementy, pokud ale chcete vědět, co vše se dá nastavit, podívejte se, prosím, do dokumentace Terracotty[36].

5.2.1 Element servers

Element *servers*, a téměř všechny jeho součásti, popíšu na základě výpisu 6. Tímto elementem specifikujeme všechny servery, které se budou nacházet v našem klastru. Pokud tedy definujeme více než jeden server (v našem případě to jsou celkově čtyři servery) a zároveň nespecifikujeme element *mirror-groups*, vytvoří všechny tyto servery pouze jednu tzv. „*mirror group*“. Já jsem však tento element uvedl a definoval jsem tak celkově dvě *mirror-group*. Do první jsem zahrnul *server1* a *server2* a do druhé potom *server3* a *server4*. Mirror groups jsou důležité, neboť právě pomocí nich Terracotta zajišťuje vysokou dostupnost a škálovatelnost. Každá tedy může obsahovat libovolný počet serverů, kde

[35]Schéma je dostupné na této adrese: http://download.terracotta.org/schema/index.html.

[36]Dokumentace je dostupná zde: http://www.terracotta.org/confluence/display/docs/Configuration+Guide+and+Reference

právě jeden z nich je tzv. „aktivní “ a ty ostatní tzv. „standby“. Aktivní server je vždy volen a stará se o požadavky klientů, respektive L1, koordinaci sdílených objektů a perzistenci dat. Standby servery plní, v rámci *mirror group*, roli záložních serverů, a neustále tak replikují veškerá sdílená data aktivního serveru. Těmito servery tedy Terracotta realizuje vysokou dostupnost, neboť kdykoliv selže aktivní server, jeden ze záložních serverů ho ihned nahradí. Pokud máme *mirror group* víc, tak jako v našem případě, dosáhneme navíc škálovatelnosti, poněvadž každý aktivní server se bude starat pouze o část sdílených dat. Mirror groups tedy navzájem komunikují a rozdělují tak mezi sebou veškerou zátěž.

```
<servers>
    <server host="%i" name="server1">
        <dso-port>9510</dso-port>
        <jmx-port>9520</jmx-port>
        <l2-group-port>9530</l2-group-port>
        <data>terracotta/server-data</data>
        <logs>terracotta/server-logs</logs>
        <statistics>terracotta/cluster-statistics</statistics>
        <dso>
            <persistence>
                <!-- Default: 'temporary-swap-only' -->
                <mode>permanent-store</mode>
            </persistence>
            <garbage-collection>
                <enabled>true</enabled>
            </garbage-collection>
        </dso>
    </server>

    <server name="server2">
        ...
    </server>

    <server name="server3">
        ...
    </server>

    <server name="server4">
        ...
    </server>

    <mirror-groups>
        <mirror-group group-name="groupA">
            <members>
                <member>server1</member>
                <member>server2</member>
            </members>
        </mirror-group>
        <mirror-group group-name="groupB">
            <members>
                <member>server3</member>
                <member>server4</member>
```

```
        </members>
      </mirror-group>
    </mirror-groups>
  </servers>
```

Výpis 6: Ukázka použití elementu *servers*

Z výpisu je vám jistě již zřejmé, že jednotlivé servery konfigurujeme elementem *server*. Údaje v něm obsažené ovšem neslouží pouze pro nastavení serveru, ale také pro všechny instance aplikace. Tímto elementem se totiž aplikace dozví, nejen jaké servery v klastru existují, ale hlavně, jak se k nim připojit. Informace, které jsou pro aplikaci důležité, jsou tak atribut *host* a element *dso-port*, popřípadě *jmx-port*. host specifikuje adresu IP či doménové jméno počítače, na kterém server běží a *dso-port* port, na kterém server naslouchá. *jmx-port* je pak vyhrazen pro monitorování a správu serveru pomocí technologie JMX. Možná se nyní ptáte, že pokud uvedeme elementů server více, jak aktuálně spouštěný server pozná, jaký *server*, respektive informace jsou pro něj určeny. K tomu je zde atribut *name*, který koresponduje s parametrem *–n*, jejž předáváme dávce pro start serveru (více později).

Elementem *data* specifikujeme adresář, do kterého bude server ukládat veškerá svá data – čili i všechny sdílené objekty při jejich perzistenci. Do *logs* budou ukládány žurnály a ve *statistics* pak najdeme výstupy, serverem provedených, statistik.

Element *dso* slouží pro definici údajů týkajících se virtuální haldy. Elementem persistence Terracottě řekneme, zda si přejeme sdílená data uchovat pouze dočasně (*temporary-swap-only*), respektive po dobu života klastru, anebo natrvalo (*permanent-store*). S *permanent-store* tedy sdílená data přežijí jak restarty serveru, tak výpadky klastru, s *temporary-swap-only* ovšem nikoliv a server v tomto módu používá pevný disk pouze jako dočasné úložiště.

V *dso* ještě můžeme nastavit distribuované garbage collection a buď ho tedy povolit, anebo úplně zakázat. I sdílená data se totiž mohou stát nepotřebnými a server tak musí vědět, zda je může odstranit či nikoliv. Sběr sdílených dat nemůže provádět aplikační JVM, protože on samozřejmě neví, zda nejsou tyto data ještě používány, v rámci klastru, i jinými JVM. Terracotta proto vyvinula distribuovaný garbage collector, který běží na každém serveru

5.2.2 Element clients

Elementem *servers* tedy konfigurujeme servery, pomocí *clients* zase nastavujeme klienty (viz výpis 7). Stejně jako u serverů, i zde můžeme uvést adresáře pro logy a statistiky, ovšem hlavním účelem tohoto elementu je specifikace tzv. „integračních modulů“. Každý integrační modul vždy definuje *tc-config.xml* pro nějakou knihovnu, a umožňuje tak její použití nad Terracotou. Jelikož dnes již existuje spoustu hojně využívaných knihoven (Spring, Hibernate apod.), existuje také mnoho integračních modulů. Ty si může každý programátor stáhnout ze stránek Terracotty[37].

[37] Konkrétně z této URL: http://forge.terracotta.org/releases/projects.html.

```
<clients>
  <logs>terracotta/client−logs</logs>
  <modules>
    <module group−id="org.terracotta.modules" name="tim−masterworker"
    version="2.2.1" />
  </modules>
</clients>
```

Výpis 7: Ukázka použití elementu *clients*

5.2.3 Element application/dso/roots

Nyní přichází na řadu jeden ze tří hlavních konceptů, na kterých je Terracotta založena. Tímto konceptem jsou tzv. „kořeny“, odtud název elementu *roots*. Obsahem tohoto elementu je libovolný počet elementů *root*, které dále obsahují element s názvem *field-name*. Tuto strukturu můžete vidět na výpisu 8, kde tedy specifikujeme celkově dva kořeny. Pomocí kořenů určujeme vše, co bude sdíleno a umístěno tak v L2.

```
<roots>
  <root>
    <field−name>org.example.TestClass.field1</field−name>
  </root>
  <root>
    <field−name>org.example2.TestClass2.field2</field−name>
  </root>
</roots>
```

Výpis 8: Ukázka použití elementu *application/dso/roots*

Kořenem můžeme být libovolný člen jakékoliv třídy, ať už statický či nikoliv, a platí, že nejen on, ale i vše, na co svými proměnným odkazuje, bude sdíleno. Tyto členy se kořeny nazývají z toho důvodu, protože vytváří tzv. „objektový graf“. Na vrcholu tohoto grafu jsou oni samotní, pod nimi jsou všechny objekty, na které mají odkaz, na další úrovni jsou objekty odkazů těchto objektů atd. Terracotta nám tedy zajistí, že celý tento objektový graf bude sdílen. Tedy i původně lokální objekt, který přidáme do tohoto grafu na jakoukoliv úroveň, se ihned stane sdíleným a L1 okamžitě odešle jeho stav do L2. Z těchto důvodů se často jako kořeny používají různé typy kolekcí.

Dále je nutné uvést, že Terracotta rozlišuje, zda je kořenem tzv. „literální “, anebo referenční typ. Mezi literální typy se neřadí pouze primitivní typy jazyka Java, ale také jejich obalovače (např. *java.lang.Integer*), enumerace, *java.lang.BigInteger*, *java.lang.BigDecimal* a také řetězce (*java.lang.String*). Pokud jako kořen uvedeme literální typ, nebude vytvářen objektový graf, ale bude sdílena pouze samotná hodnota.

Druhým a posledním rozdílem mezi referenčními a literálními typy je fakt, že hodnota referenčního kořene se po vytvoření již nemůže měnit. U literálních kořenů to ovšem neplatí, ty může, v průběhu aplikace, nastavovat opakovaně.

Poslední zvláštností kořenů je jejich životnost. Kořen totiž žije po dobu života celého klastru a to i v případě, že již skončily všechny instance aplikace. Není tak na něj aplikován distribuovaný garbage collector, čímž se liší od všech ostatních objektů objektového grafu, které skončení poslední instance aplikace nepřežijí.

5.2.4 Element application/dso/instrumented-classes

V kapitole o aplikaci *HelloClusteredWorld* jsem již nakousl, že Terracotta provádí instrumentaci, tedy modifikaci bytecode tříd. To provádí z toho důvodu, aby vůbec realizovala možnost určité objekty sdílet. Je důležité říci, že se modifikují pouze ty třídy, které jakýmkoliv způsobem přijdou do styku se sdílenými objekty. Modifikovat se musí proto, jelikož instrukce, které se sdílenými objekty pracují, musí být obohaceny o instrukce, které komunikují s L2.

Je důležité, že instrukce jsou modifikovány tak, aby v případě, že sdílený objekt, který doposud není v L2, byl do L2 umístěn (tato situace nastane v případě, že aktuální instance aplikace je vůbec první instancí, která se sdíleným objektem pracuje). Pokud se však již sdílený objekt v L2 nachází, jeho hodnotu již instance aplikace, respektive L1 nevytváří, nýbrž pouze stahuje z L2. Toto vše je samozřejmě prováděno pouze v případě operátoru new čili při vytváření sdíleného objektu. Pokud se sdílený objekt jenom čte či nastavuje, nemusí se žádná kontrola, existence objektu v L2, vykonat.

Třídy, které mají být instrumentovány, specifikujeme pomocí elementu *instrumented-classes*. To znamená, že třídy k instrumentaci nejsou Terracottou nijak automaticky hledány, ale všechny je musí programátor znát. Je jasné, že pokud nějakou třídu, která musí být v dané aplikaci modifikována, v *instrumented-classes* neuvedeme, aplikace se nebude chovat korektně.

Na výpisu 9 je ukázáno, jakým způsobem se specifikují třídy k modifikaci (element *include*), respektive ty třídy, které instrumentovány být nemají (element *exclude*). Třídy určujeme výrazy jazyka AspectWerkz[38], který se ostatně také využívá pro určení kořenů a zámků (viz element *locks*, který je popisován dále v textu).

```
<instrumented-classes>
  <include>
    <class-expression>org.example.*</class-expression>
  </include>
  <include>
    <class-expression>org.example2.*</class-expression>
  </include>
  <exclude>org.example.test.*</exclude>
  <exclude>org.example.test2.*</exclude>
</instrumented-classes>
```

Výpis 9: Ukázka použití elementu *application/dso/instrumented-classes*

[38] Domovská stránka projektu AspectWerkz je přístupná pod touto adresou: http://aspectwerkz.codehaus.org/.

Další věc, která se k instrumentaci váže, je způsob, jakým Terracotta modifikuje bytecode JavaSE tříd. To jsou vesměs třídy ze souboru *rt.jar*, a jak možná víte, bytecode těchto tříd již nelze měnit po startu JVM. Jak to, že ale můžeme sdílet např. objekty typu *java.lang.String* nebo *java.lang.Integer*, když jsem řekl, že veškeré třídy, které přijdou do styku se sdílenými objekty, musí být modifikovány? To je dáno tím, že Terracotta využívá parametr *–Xbootclasspath*[39] programu *java*, a může tak instrumentovat i třídy z *rt.jar*.

5.2.5 Element application/dso/transient-fields

Pokud je objekt sdílen, je jeho stav, čili hodnoty všech jeho členů, vždy přenášen do L2. Co když ale určitý člen sdílet nechceme? Přesně k tomuto účelu byl vytvořen element *transient-fields*, kterým specifikujeme, jaké členy, jinak sdílených tříd, sdílet nechceme. Jeho použití lze vidět na výpisu 10.

```
<transient−fields>
   <field−name>org.example.MyClass1.fieldA</field−name>
   <field−name>org.example2.MyClass2.fieldB</field−name>
   <field−name>org.example2.test.MyClass3.fieldB</field−name>
</transient−fields>
```

Výpis 10: Ukázka použití elementu *application/dso/transient-fields*

5.2.6 Element application/dso/locks

Po kořenech a instrumentaci jsou zámky (*locks*) třetím a posledním hlavním konceptem Terracotty. Zámky jsou velmi důležité, neboť práce, s libovolným sdíleným objektem, musí vždy probíhat v rámci tzv. „distribuovaného zámku“. Distribuovaný zámek je klasický Javovský zámek, jehož platnost je rozšířena do celého klastru. Platí tudíž pro všechny klientské JVM. Pokud bychom tedy ke sdílenému objektu přistupovali mimo kontext libovolného distribuovaného zámku, Terracotta vyhodí vyjímku. Toto je tedy nová sémantika, kterou musí programátor Terracotta aplikací znát. Liší se tak od chování „běžného“ JVM, neboť JVM nikdy nevyhazuje výjimku z důvodu absence zámku.

Distribuované zámky jsou Terracottou automaticky vytvářeny tehdy, pokud:

- Vstoupíme do synchronizovaného bloku, jehož parametrem je libovolný sdílený objekt.
- Vstoupíme do synchronizované metody, kterou voláme na aktuálně sdíleném objektu.
- Použijeme instanci *java.util.concurrent.locks.ReentrantReadWriteLock*, která je sdílena.

Distribuované zámky nám tak zaručí exkluzivní přístup i ke sdíleným objektům. Nicméně pokud nám stačí pouze zámek pro čtení, tedy nepožadujeme exkluzivní přístup

[39] Viz například toto URL: http://java.sun.com/javase/6/docs/technotes/tools/windows/java.html

ke sdíleným objektům, Terracotta nám toto taktéž povolí. V Terracottě tudíž existuje více typů zámků, přičemž tři nejdůležitější jsou tyto:

- *write* – výchozí zámek pro získání exkluzivního přístupu
- *read* – sdílený zámek pro simultánní čtení
- *synchronous-write* – zámek *write* ovšem s tím rozdílem, že L1 ho neodebere vlastníku do doby, než všechny provedené změny, na sdílených objektech, neodešle do L2 a ten tyto změny nepotvrdí

Doposud jsem se ještě nezmínil, kdy Terracotta zasílá provedené změny, na sdílených objektech, do L2. Terracotta zavádí pojem tzv. „transakce“, která obsahuje vždy ty změny, které byly provedeny v rámci jednoho získání a uvolnění distribuovaného zámku. Tyto transakce pak L1, pro optimalizaci, sdružuje do dávek a až ty odesílá do L2. Toto chování ovšem výrazně ovlivňuje paměťový model Javy, který je Terracotta nucena dodržovat. Pokud tedy určité změny musí být, dle paměťového modelu, viditelné v danou chvíli i ostatním instancím, Terracotta žádnou optimalizaci neprovede a změny do L2 ihned odešle. Ten je pak obratem zasílá do všech L1, které tyto změny aktuálně potřebují. Těmito L1 jsou ty, které v danou chvíli pracují s takovými sdílenými objekty, jež byly změněny.

Na výpisu 11 můžete vidět, jakým způsobem se zámky v Terracottě konfigurují. Elementem *autolock*, respektive *method-expression* vždy určíme metodu, ve/na které bude Terracotta, při startu aplikace, hledat klíčová slova *synchronized* a instance *ReentrantReadWriteLock*. Pokud je nalezne, bytecode této metody modifikuje tak, aby zaručila chování, které jsem zde popsal. Konfigurace zámků tedy není automatická a programátor tak musí sám vědět, ve kterých metodách je nucen požádat o vytvoření, respektive uvolnění distribuovaných zámků.

```
<locks>
   <autolock>
      <method-expression>* org.example.MyClass1.set*(..)</method-expression>
      <lock-level>write</lock-level>
   </autolock>
   <autolock>
      <method-expression>* org.example.MyClass1.get*(..)</method-expression>
      <lock-level>read</lock-level>
   </autolock>
</locks>
```

Výpis 11: Ukázka použití elementu *application/dso/locks*

5.2.7 Element application/dso/distributed-methods

Posledním elementem, který vám zde popíši, je tag *distributed-methods*. Ten je velmi zajímavý, neboť jsme, s jeho pomocí, schopni vytvořit tzv. „distribuované volání metod“

neboli DMI. Kdykoliv tedy určitá instance aplikace zavolá distribuovanou metodu, je tato metoda ihned provedena lokální instancí a zároveň jsou notifikovány všechny ostatní instance, aby metodu také provedly.

Distribuovanými metodami tak s výhodou např. naimplementujeme distribuovanou variantu vzoru Event/Listener, a můžeme tak klastrovat např. „Swingovské“ aplikace.

Avšak pozor, DMI není vhodné pro synchronizaci, neboť sama Terracotta říká, že volající nemá žádnou záruku, kdy přesně se volaná metoda na ostatních JVM zavolá.

Ukázku konfigurace DMI ilustruje výpis 12.

```
<distributed-methods>
  <method-expression>
    void org.example.MyClass.somethingHappened(String, int)
  </method-expression>
  <method-expression>
    String [] org.example2.AnotherClass.eventOccurred(Boolean, Double)
  </method-expression>
</distributed-methods>
```

Výpis 12: Ukázka použití elementu *application/dso/distributed-methods*

5.3 Spouštění Terracotta aplikací

Na přiložené CD jsem umístil výše popsanou *HelloClusteredWorld* aplikaci, na které vám nyní popíši, jak ji spustíme nad Terracottou. Veškeré spouštění probíhá prostřednictvím dávek, které jsou obsaženy v distribuci Terracotty[40].

Pro spuštění serveru se přepněte do složky s aplikací a zadejte příkaz *%TC_HOME%/start-tc-server.bat -f tc-config.xml*, kde řetězec *%TC_HOME%* specifikuje kořenovou složku distribuce. Pokud jste uživatelem UNIXu, namísto *.bat* zadejte *.sh*.

Server již máme spuštěn, nyní nezbývá než spustit jednotlivé instance aplikace. To provedeme opět velmi jednoduše, stačí, když otevřeme požadovaný počet konzolí, přepneme se v nich opět do složky s aplikací a zadáme příkaz *%TC_HOME%/dso-java.bat HelloClusteredWorld*.

[40] Distribuci Terracotty si můžete stáhnout z této adresy: http://www.terracotta.org/dl/oss-download-catalog. Dávky jsou umístěny ve složce *bin*.

6 Praktické využití Terracotty

Stěžejním bodem této práce bylo provést analýzu a návrh vlastní aplikace, která by ukázala praktické využití Terracotty při řešení nějakého reálného problému. Dále vyřešit zvolený problém i bez pomoci Terracotty a následně tato dvě řešení porovnat nejen s ohledem na jejich výkonnostní parametry, ale také jak složité je bylo navrhnout a vytvořit.

Již víme, že Terracotta se dá použít k mnoha účelům jako distribuované cachování, replikace relací či změna relačního modelu na objektový. Možná si ještě pamatujete, že ale tím nejtypičtějším případem užití je tzv. „workload partitioning“. Doslova tento pojem znamená rozdělení (výpočetní) zátěže a možná ho spíše znáte pod populárnějšími názvy jako MapReduce[41], data grid či compute grid. Používá se k řešení výpočetně náročných úkolů jako je dotazování či modifikace velkého objemu dat a jeho hlavní myšlenka spočívá v tzv. paradigma „rozděl a panuj“. Jednoduše řečeno, řešení každého složitého problému spočívá v jeho rekurzivním rozdělení na takové podproblémy, které jsou již dostatečně jednoduché na to, aby mohly být již přímo vyřešeny. Workload partitioning tedy spočívá v rozdělení (výpočetního) úkolu (tzv. job či task) na tzv. „work items“, což jsou jednotky práce, které mohou být vykonávány současně.

Jeden z nejznámějších a nejpoužívanějších (architektonických) vzorů pro paralelizaci práce je tzv. „Master/Worker“. A poněvadž je tento vzor, respektive workload partitioning, typickým případem užití Terracotty, právě proto jsem si ho záměrně zvolil pro návrh a realizaci řešení, mnou zvoleného, výpočetního problému (popisuji dále v textu). Praktická část mé práce tedy spočívá v realizaci tohoto vzoru nad Terracottou a bez ní a nalezení takového problému, jehož řešení lze dobře paralelizovat.

Master/Worker je především klíčovým vzorem v tzv. grid computing a to dokonce do takové míry, že většina „grid“ technologií je de facto jeho průmyslovou implementací. Pro jeho snadnější pochopení, respektive získání lepší představy o workload partitioningu, mi, prosím, dovolte, se nejprve letmo věnovat této oblasti.

6.1 Co jsou gridy?

Gridy jsou specializované distribuované systémy, jejichž počítače (uzly) vytváří infrastrukturu o síle superpočítače a slouží pro správu a vykonávání výpočetně náročných úkolů. Tyto úkoly mohou být nastartovány, zastaveny či pozastaveny a pro dosažení větší škálovatelnosti a vyššího výkonu mohou být jejich data i operace přenášena z uzlu na uzel. Škálovatelnosti dosahují pomocí lokality referencí (viz níže) a dostupnosti prostřednictvím efektivní duplikace dat.

Ačkoliv existuje více typů gridů, nejběžnější jsou gridy datové. Datové gridy totiž dnes představují velmi zajímavou službu pro enterprise aplikace, neboť jim automaticky poskytují „failover“ a „fault tolerance“. Aplikační data jsou v datovém gridu uložena v pamětích jeho jednotlivých počítačů a tím je značně sníženo zpoždění k jejich přístupu. Pokud bychom tak měli např. 10 serverů, tak každý uzel by se v datovém gridu staral o jednu desetinu aplikačních dat.

[41] Více informací viz http://en.wikipedia.org/wiki/MapReduce.

6.1.1 Jak gridy zachází se škálovatelností?

Jedním z hlavních důvodů, proč gridy škálují tak dobře je, že mohou dělat inteligentní rozhodnutí, zda přesunout buď data do jejich „zpracovávajícího“ kontextu, anebo raději „zpracovávající “ kontext k datům. Tak jako každý objekt, i výpočetní úkol se totiž skládá vždy z dat a operací, které chceme nad těmito daty provést. Tato data mohou být v gridu uložena kdekoliv, stejně tak jako tyto operace mohou být vykonávány jakýmkoliv jeho uzlem. V každém případě je ale žádoucí, aby operace pracovaly pouze s daty, které jsou danému uzlu lokální. Tato lokalita referencí totiž minimalizuje zpoždění a umožňuje tak téměř neomezenou škálu. Ne vždy je ovšem možné, aby uzel pracoval pouze s daty lokálními, neboť toto je dáno především povahou daného problému. Čím více tedy daný problém obsahuje sdílených dat, tím častěji dochází k jejich přesunu a potřebě synchronizace. Absence lokality referencí tak nutně vede ke snížené škálovatelnosti a řešení jakéhokoliv problému by mělo být tedy navrženo tak, aby jeho jednotlivé „work items“ mohly pracovat co možná nejvíce nezávisle na sobě.

Ideálním případem jsou tzv. „embarrassingly parallel“[42] problémy, jejichž rozdělení na jednotlivé paralelní úkoly (work items) nedá téměř žádnou práci. Každý takovýto work item obsahuje totiž vše, co potřebuje a s ostatními nesdílí žádný sdílený stav. Tyto jednotky práce tak mohou být spouštěny v absolutní izolaci a zcela nezávisle na ostatních. I já jsem si pro mé řešení vybral problém tohoto typu, ale o tom až později.

6.1.2 Jak gridy zachází s failover a vysokou dostupností?

Odolnost gridů proti výpadkům a chybám všeho druhu spočívá ve dvojnásobném či trojnásobném kopírování aplikačních dat mezi jeho jednotlivými uzly.

Zotavení z chyby probíhá pro aplikaci zcela transparentně, neboť o chyby typu výpadek uzlu, selhání work item, chyba sítě apod. se infrastruktura gridu automaticky postará. Např. work items, které měly být vykonány vypadnuvším uzlem, mohou být přesměrovány na jiný uzel nebo může být uzel restartován.

6.1.3 Případy užití

Aplikace, které mohou těžit z výhod, které jim gridy nabízí, jsou obvykle programy, které pracují s velkými objemy dat. Tyto aplikace potřebují zpracování dat urychlit a gridy jim toto umožňují formou paralelizace.

Gridy tedy především slouží pro výpočty složitých vědeckých a matematický problémů, používají se k předpovědi stavu ekonomiky, seizmické analýze anebo k vyhledávání v Google či Yahoo! indexech.

[42] Význam tohoto slovního spojení můžete nalézt například zde: http://en.wikipedia.org/wiki/Embarrassingly_parallel.

6.2 Architektonický vzor Master/Worker

Již víme, že tento architektonický vzor je nejběžnějším vzorem pro paralelizaci práce. I přes to, že je velice oblíben, není vůbec složitý, neboť se skládá z pouhých tří entit: mistra (Master), sdíleného (pracovního) prostoru a jednoho či více dělníků (Worker). Sdílený prostor je obvykle realizován nějakou FIFO[43] datovou strukturou, ale může být také implementován jako tzv. „tuple space“[44].

Životní cyklus výpočtu je jednoduchý a řídí ho kompletně Master. Nejprve jsou vytvořeny jednotlivé work items, které představují řešení daného problému. Ty jsou následně mistrem umístěny do sdíleného prostoru, z něhož si je postupně vyzvedávají jednotliví dělníci. Dělníci tyto jednotky práce poté vykonávají a po jejich (úspěšném) dokončení zasílají vygenerované dílčí výsledky zpět mistrovi. Mistr po dobu výpočtu vždy čeká i na ten poslední dílčí výsledek a až poté, co ho obdrží, může výpočet ukončit.

Jedním z hlavních důvodů, proč tato jednoduchá koncepce pro paralelizaci práce dostačuje, je, že tento algoritmus automaticky rozděluje danou zátěž rovnoměrně. Tuto rovnováhu umožňuje právě ten sdílený pracovní prostor, neboť jen tak mohou dělníci okamžitě po dokončení jednoho úkolu získat z prostoru další jednotku a pokračovat tak v práci bez jakéhokoliv přerušení. Důležité je, že dělník nijak nerozlišuje mezi jednotlivými prácemi a jeho činnost končí pouze v případě, že v prostoru již žádná práce nezbyla. Tato architektura obvykle vykazuje dobrou škálovatelnost v případě, že množství práce vysoce přesahuje počet dělníků, a když každá práce potřebuje ke svému dokončení přibližně stejný čas. Pro lepší představu algoritmu se, prosím, podívejte na obrázek 10.

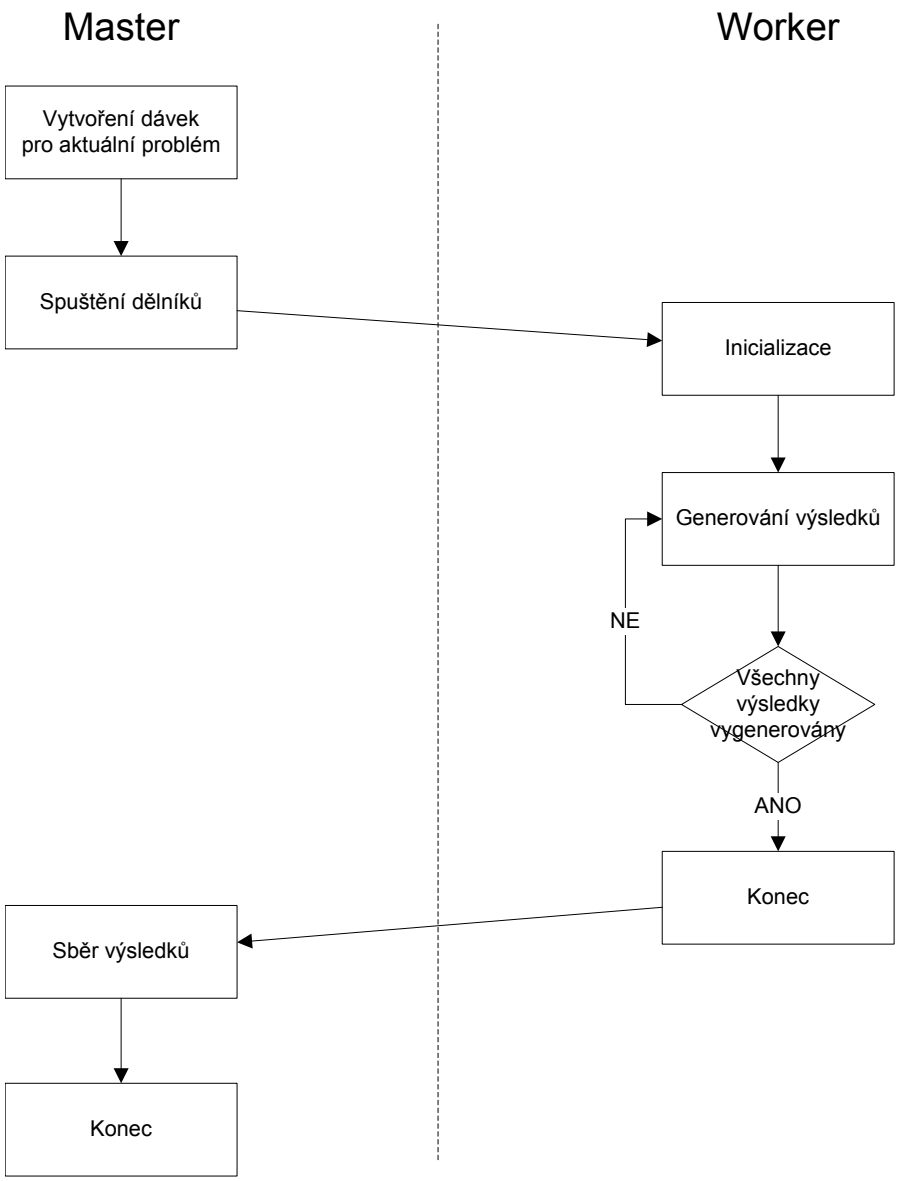
Již jsem říkal, že praktickou část mé práce jsem zasvětil tomuto architektonickému vzoru. Dal jsem si za cíl realizovat ho nad Terracottou a také bez ní a poté tyto řešení ověřit na nějakém reálném problému. A poněvadž se Terracotta týká distribuovaných systémů, mým úkolem bylo tedy realizovat distribuovanou variantu tohoto vzoru. Stěžejní podmínkou obou architektur tak bylo umožnit spouštět dělníky i na jiných počítačích, než na kterém běží mistr.

Ačkoliv se tedy může implementace vzoru v té nejjednodušší podobě zdát velice jednoduchá – vystačili bychom si totiž s obyčejnou kolekcí jazyka Java a jejími synchronizačními primitivy – stěžejní podmínku by toto řešení nesplňovalo, neboť by fungovalo pouze v rámci jednoho JVM. Distribuovaná varianta již přináší bohužel řadu problémů, a aby byla dobře využitelná i v praxi, musí splňovat několik podmínek. Tyto podmínky popisuji později, ale platí, že jakákoliv průmyslová implementace by je měla splňovat.

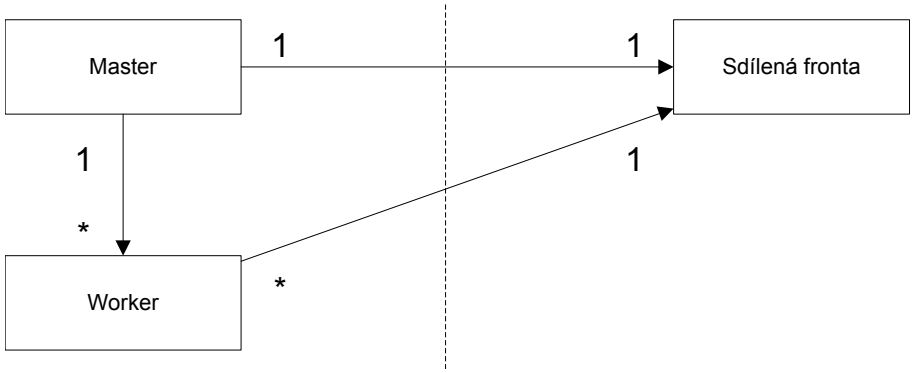
Jazyk Java, respektive platforma JavaSE od verze 1.5, nám již nabízí přímou podporu tohoto architektonického vzoru. Tato podpora má podobu rozhraní *ExecutorService* a ostatních podpůrných *java.util.concurrent* tříd a ačkoliv je jeho výchozí implementace opět platná pouze v rámci jednoho JVM, jeho popis je důležitý ze dvou důvodů: za prvé, Terracotta totiž nabízí jeho distribuovanou variantu, kterou jsem nakonec využil ve svém projektu i já a za druhé, uvědomění si jeho nedostatků nám lépe pomůže porozumět

[43] FIFO datové struktury se chovají jako klasická fronta. Tedy, kdo dřív přijde, ten je také dříve obsloužen.

[44] Tuple space je implementace asociativní paměti pro účely paralelních, respektive distribuovaných výpočtů. Ukládají se do ní tzv. tuples neboli n-tice, ke kterým můžeme přistupovat paralelně. Více si můžete přečíst např. zde: http://en.wikipedia.org/wiki/Tuple_space.



Obrázek 10: Algoritmus vzoru Master/Worker [1]



Obrázek 11: Architektura Master/Worker systému při využití *ExecutorService* [1]

podmínkám, respektive vlastnostem, které by měly všechny komerční implementace tohoto vzoru mít.

6.2.1 Rozhraní ExecutorService

Toto rozhraní představuje mistra a thread pool (množina vláken) jeho implementací jednotlivé dělníky. Architektura této služby je jednoduchá, ale i přesto velmi schopná, neboť, jak můžete vidět na obrázku 11, programátor implementuje kompletní Master/Worker pomocí jediné abstrakce a frameworku. Různorodé implementace tohoto rozhraní nám nabízí tovární metody pomocné třídy *Executors*, ale i tak všechny tyto implementace spojuje totožná architektura z obrázku – tzn., že všechna vlákna, respektive dělníci, vždy přistupují k jediné sdílené frontě (zpravidla k instancím typu *BlockingQueue*).

Pokud se podíváte na rozhraní této služby[45], můžete vidět, že v něm chybí metody pro zahájení a ukončení výpočtu. To je ovšem v pořádku, poněvadž každý mistr je vždy specifický danému problému. *ExecutorService* tak představuje pouze framework pro jejich implementaci a mistr potom, co vytvoří jednotlivé work items, umožní dělníkům jejich spuštění prostřednictvím metod *submit*. Návratová hodnota těchto metod je velmi důležitá, neboť mistrovi umožňuje správu životního cyklu dané jednotky práce a také možnost synchronně si počkat na její výsledek. Ostatní metody nejsou z hlediska našeho vzoru až tak důležité a pro jejich pochopení si proto, prosím, v případě zájmu prostudujte Javadoc.

Říkal jsem, že ačkoliv je tato služba ve většině případů dostačující, z mého „distribuovaného“ pohledu již nikoliv. Prvním nedostatkem totiž je, že její rozhraní nijak neodděluje mistra od dělníků. Oba dva jsou tak součástí pouze jediné abstrakce, a proto je nemožné řídit mistra nezávisle na dělnících. V praxi tak tento nedostatek znamená, že služba je použitelná pouze v rámci jediného JVM.

Druhým a snad i vážnějším problémem je absence jakékoliv vrstvy kontroly. *ExecutorService* je totiž implementováno jako černá skříňka, a tak se programátor nemůže jak

[45]To je dostupné například na této adrese: http://java.sun.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html.

dozvědět, zda jednotka práce byla spuštěna, pozastavena či zamítnuta v důsledku nějaké chyby. Díky tomu tak vývojář nemůže zareagovat na nastalou chybu např. opakovaným spuštěním work item na stejném či úplně jiném výpočetním uzlu.

Těmito nedostatky nemůže trpět samozřejmě žádné komerčně použitelné řešení a jako takové by navíc mělo splňovat, již slíbené, následující podmínky:

- Schopnost zvládat i enormní objemy dat - již víme, že sdílený pracovní prostor *ExecutorService* architektury je realizován pouze jedinou frontou (viz obrázek 11). Tato fronta by se ovšem, v distribuovaném prostředí, záhy ukázala jako velmi vážný výkonnostní problém, neboť by o ní najednou soupeřili všichni dělníci z celého klastru. A poněvadž tedy v daný okamžik může s frontou pracovat pouze jediné vlákno nebo proces, mnoho dělníků by tak muselo zbytečně čekat a nemohlo provádět svou práci. Pouze jediná fronta tak není vhodným řešením a často jsou tak v praxi mezi dělníkem a mistrem umístěny celkově dvě dedikované fronty. Každý dělník tak vlastní dva ryze soukromé komunikační kanály, kde jedním přijímá jednotky práce a druhým zasílá mistrovi zpět výsledky. Tato architektura mnoha front tak umožňuje zvládnutí i opravdu velkého objemu dat, poněvadž zde již nedochází ke zbytečným prodlevám.

- Podpora směrování - díky tomu, že zde již není pouze jediná fronta, ale každý dělník má svou, musí se nyní mistr vždy rozhodnout, do jaké fronty, čili kterému dělníku, příchozí práci zašle. Právě tomuto inteligentnímu výběru té nejvýhodnější fronty se říká směrování a danému programu směrovací algoritmus. Se směrováním se např. setkáváme také na třetí vrstvě ISO/OSI[46] modelu a typickými směrovacími algoritmy jsou Round-Robin[47] či klasický „load balancing“[48].

- Odolnost vůči chybám jednotek práce - každá implementace Master/Worker systému by měla počítat s možným výskytem chyby při vykonávání jednotky práce. V ideálním případě by tak měla zcela automaticky, a pro aplikaci naprosto transparentně, na tyto výjimečné situace reagovat např. opakovaným spouštěním. Toto ovšem není úplně nutné a tak jako minimum se alespoň jeví schopnost oznámit nastalou chybu volající aplikaci.

- Imunita vůči selhání dělníků - stejně jako v případě work items, i na výpadky dělníků musí být systém připraven. Zde je ale situace poměrně složitější, neboť pro správnou činnost systému musí být všechny nedokončené práce dělníka buď automaticky přesměrovány do jiné pracovní fronty, nebo opět oznámeny, ve formě nějaké události, uživateli. Praxe navíc ukázala, že již samotná detekce výpadku může být obtížná.

[46] Čili na úrovni protokolu IP. Více se o OSI modelu dozvíte například zde: http://en.wikipedia.org/wiki/Iso_osi.

[47] Algoritmus Round-robin si všechny plánované prvky uchovává v jedné frontě a při výběru vybere vždy ten prvek, který se aktuálně nachází na jejím vrcholu. Po výběru jej z vrcholu zase odebere a umístí úplně nakonec.

[48] Load balancing algoritmy provádí výběr dle aktuálních hodnot určitých sledovaných parametrů. V počítačových sítích se tak např. rozhodují na základě aktuální zátěže, propustnosti linky, jejího stavu apod.

- Dynamická správa klastru - poslední podmínka umožňuje dělníkům a mistrům připojovat se k a odpojovat se od gridu za běhu.

Na závěr těchto vlastností a podmínek by chtěl ještě zmínit jednu velmi starou, ale praxí osvědčenou techniku, která může velice výrazně urychlit celý výpočet daného problému. Tou technikou je tzv. „bulk-loading“ neboli dávkové zpracování a jeho myšlenka je velice jednoduchá. V kontextu Master/Worker systému totiž pro nás znamená pouze neposílat jednotlivé jednotky práce dělníkům postupně, ale raději ve větších skupinách, tedy dávkách. Tím, že dělník nezíská najednou ze své pracovní fronty pouze jednu jednotku, ale pokaždé jejich větší množství, se výrazně sníží počet přístupů k této frontě. Pokud i výsledky těchto prací nebudeme posílat jednotlivě, ale vždy najednou, tak tímto celkovým snížením počtu přístupů docílíme enormního zrychlení, neboť již zde nebude docházet k tak časté synchronizaci.

6.3 RayTracer

Poté, co již víme, jaké podmínky budou muset má dvě řešení (tj. s pomocí Terracotty a bez ní) Master/Worker systému splňovat, vám nyní mohu konečně představit problém, na jakém tyto implementace následně otestuji.

Jak již anglický název podkapitoly napovídá, rozhodl jsem se pro algoritmus výpočtu globálního osvětlení 3D scény metodou sledování paprsku[49]. Tím problémem a zároveň požadovaným výstupem je zde tedy zhotovení digitálního obrázku, který vznikne vyrenderováním nějaké scény. Tento algoritmus jsem si vybral záměrně, neboť ve své nejjednodušší, tj. neoptimalizované, podobě je jeho řešení „embarrassingly parallel“ (viz předchozí text). Barvu každého bodu obrazu tak lze spočítat zcela nezávisle na ostatních, a tak je tato jeho implementace naprosto ideální pro běh v nějakém „grid“, potažmo Master/Worker systému.

Realizace ray tracing enginu ovšem není vůbec jednoduchá, a proto se můžeme setkat s jeho jak sofistikovanými, tak velice jednoduchými implementacemi. Tato řešení se zpravidla odlišují stupněm prováděných optimalizací a především pak podporou různých pokročilých technologií[50].

Z těchto důvodů jsem nerealizoval engine sám, ale použil jsem raději již hotový projekt[51], který je samozřejmě embarrassingly parallel. Tento projekt je poměrně jednoduchý a podporuje pouze tři základní typy světel[52], mapování textury a jediný grafický objekt v podobě koule. Ve svém projektu jsem ale musel ještě toto řešení připravit na koncept mistra a dělníků, a proto jsem provedl také jeho refaktorizaci. Výslednou podobu tříd, jejich názvů a umístění do jednotlivých balíčků ale popíši až později ve fázi návrhu.

[49] Tedy pro algoritmus, který se nazývá ray tracing. Pro více informací o tomto algoritmu se, prosím, podívejte například zde: http://en.wikipedia.org/wiki/Ray_tracing_(graphics).

[50] Např. HDR (http://en.wikipedia.org/wiki/High_dynamic_range_imaging), Depth of Field (http://en.wikipedia.org/wiki/Depth_of_Field) nebo Subsurface scattering (http://en.wikipedia.org/wiki/Subsurface_scattering).

[51] Využil jsem implementaci od pana Dean Camera, která je dostupná na jeho blogu z této URL: http://www.fourwalledcubicle.com/RayTracer.php.

[52] Flood Light, Omni Light a Spot Light.

6.3.1 Vize

Mým cílem je vytvořit aplikaci, která uživateli umožní vyrenderovat jím vybranou scénu. Ta bude zadávána ve formě XML a navíc si bude možné nastavit parametry její grafické projekce. Zodpovědnost za vlastní rendering budou mít v aplikaci jednotlivé implementace Master/Worker systému a uživatel si tak před započetím vykreslování vždy jednu vybere. Každá tato implementace by měla také splňovat podmínky minulé kapitoly, především pak schopnost poradit si s výpadky dělníků. Typicky se zde bude používat též dávkové zpracování, a tak velikost dávky, případně i ostatní parametry specifické vždy dané implementaci, by měly jít také nastavit. Dále, sledování paprsku je výpočetně náročné, a tak bude navíc uživatel schopen renderovací proces, kdykoliv v jeho průběhu, zastavit.

Rozhraní aplikace bude realizováno pomocí grafického frameworku JFC[53], a aby toto rozhraní bylo za všech okolností vždy aktivní, vykreslování nebude moci být prováděno na EDT[54]. Program bude uživatele informovat také o stavu aktuálního renderingu a to prostřednictvím chybové konzole, informačních hlášek, uplynulého času a procentuálního vyjádření počtu již vyrenderovaných pixelů.

Aplikaci jsem nazval „*RayTracer*“.

6.3.2 Funkční a nefunkční požadavky

Podrobná specifikace funkčních a nefunkčních požadavků je uvedena v příloze A, zde uvádím pouze souhrnný use-case diagram (viz obrázek 12.). Pro specifikaci požadavků byl využit model FURPS+, přičemž funkční požadavky byly popsány dle doporučení z [3].

6.3.3 Iterativní vývoj aplikace

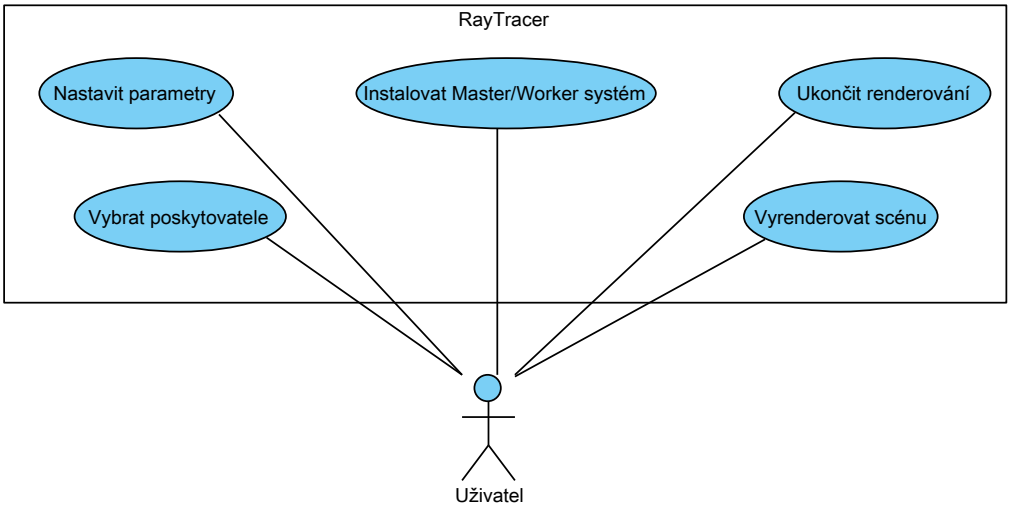
Aplikaci realizuji iterativním přístupem a z tohoto důvodu jsem si stanovil tyto iterace:

1. Nejprve se budu zabývat uživatelským rozhraním, a tak výstupem první iterace bude již téměř finální podoba aplikace. Aplikaci tak půjde již spustit, ukončit, do jejího návrhu budou dále zakomponovány všechny relevantní nefunkční požadavky a navíc bude možné pracovat již s parametry.

2. V druhé iteraci provedu refaktorizaci využité implementace ray tracer algoritmu a připravím ji tak pro běh v gridu. Následně navrhnu a implementuji komunikační rozhraní s Master/Worker systémy a navíc uvedu procesní pohled na celkovou architekturu systému.

[53] Tedy převážně pomocí technologie Swing, což je jedna ze tří částí frameworku JFC. Zbylými částmi jsou AWT a Java 2D.

[54] EDT je vlákno, které jako jediné aktualizuje vzhled grafických aplikací využívajících AWT. Jeho účelem je tedy přijímat a následně vykonávat požadavky na překreslení určitých částí GUI, tak zpracovávat vstup ze zařízení jako je myš nebo klávesnice. Je tedy velmi důležité, aby aplikace na tomto vlákně nevykonávala žádné déle trvající úlohy, neboť by měly za důsledek „zamrznutí“ UI.



Obrázek 12: Funkční požadavky na aplikaci *RayTracer*

3. Po třetí iteraci bych měl být schopen již aplikaci kompletně otestovat, neboť v ní provedu návrh a realizaci prvního Master/Worker systému. Ten, ačkoliv nebude splňovat nefunkční požadavky, neboť nebude distribuován, a bude tak fungovat pouze v rámci jediného JVM, bude pro další fázi vývoje důležitý, neboť na něm ukážu, jak jednoduše můžeme, pomocí Terracotty, toto nedistribuované řešení rozšířit na libovolný počet JVM.

4. Čtvrtá fáze bude stěžejní iterací této diplomové práce, neboť v ní realizuji koncept mistra a dělníků nad Terracottou. Na reálném příkladu tak konečně ukážu nejenom to, o čem jsem v teoretické části stále mluvil, čili škálovatelnost a vysokou dostupnost, ale také to, jak jsem již ostatně řekl, jak jednoduše, a to opravdu téměř s žádnými změnami, můžeme udělat z řešení třetí iterace velmi výkonný a robustní distribuovaný systém pro renderování scény.

5. V páté iteraci konečně zhotovím Master/Worker systém bez Terracotty a budu přitom sledovat, o kolik složitější, to bez Terracotty je.

6. V poslední iteraci dokončím vše, co jsem ještě nestihl a sepíšu tak např. uživatelskou a programátorskou dokumentaci. Nakonec přidám podporu pro instalaci modulů a načítání scény ze souboru.

6.3.4 První iterace

V nefunkčních požadavcích jsem si stanovil, že aplikace *RayTracer* musí být napsána v jazyce Java a mít grafické uživatelské rozhraní. Grafika v Javě to je synonymum pro třídy JFC, a proto ji realizuji pomocí frameworku Swing. Ačkoliv je ale Swing velmi mocná a obsáhlá knihovna, jistě mi dáte za pravdu, že mnoho problémů, se kterými se vývojáři

GUI aplikací potýkají každý den, bohužel neřeší. Chybí zde tak např. jednoduchá správa životního cyklu, snazší práce s akcemi, vlákny, zdroji a jejich lokalizací a také perzistence.

Z tohoto důvodu využiji ještě tzv. Swing Application Framework[55], což je sice poměrně útlá kolekce Java tříd, které ale i tak poskytuje pro většinu desktopových aplikací zcela dostatečnou infrastrukturu. Framework jako takový tedy nabízí:

- Životní cyklus aplikace - zejména její inicializaci, spuštění a následné ukončení
- Podporu pro správu a nahrávání zdrojů jakými jsou řetězce, formátované zprávy, obrázky, barvy, fonty a všechny ostatní typy společné všem GUI aplikacím
- Podporu pro definici, správu a svázání jak synchronních, tak asynchronních akcí k jednotlivým GUI komponentám
- Perzistentní aplikační stav, respektive podporu pro automatické či manuální ukládání stavu GUI

Architektura frameworku je velice jednoduchá, a tak rozhraní mé aplikace bude složeno pouze z jediné třídy – Application (viz obrázek 3 v příloze C). Bude tvořeno „swingovskými“ komponentami (*JComponent*) a rozšíří třídu *SingleFrameApplication*. Tato dědičnost nám zaručí přístup k metodám životního cyklu a také ke všem službám, jaké Swing Application Framework nabízí. Ty jsou dostupné prostřednictvím aplikačního kontextu (*ApplicationContext*).

RayTracer bude využívat celkově tři služby a to *LocalStorage* pro trvalé uložení parametrů, *ResourceManager* pro práci se zdroji a *ActionManager* k obsluze událostí.

6.3.4.1 Výstup iterace Obrázek 1 v příloze B ukazuje finální podobu aplikace a snad je na něm i patrné, že jsem se snažil dodržet i požadavky na estetiku a konzistenci UI v programu. Do jeho horní části jsem umístil menu a nástrojovou lištu, jejichž prostřednictvím se uživatel dostane jak k nastavení programu (*Tools/Settings*), tak chybovému žurnálu (*Tools/Error Log*), vybere si poskytovatele Master/Worker systému a spustí, popřípadě zastaví jednotlivá renderování. Scéna pak bude vykreslována do jeho prostřední části a stavová lišta následně ukáže uplynulý čas a počet již vyrenderovaných pixelů v procentech.

Nastavení parametrů uskuteční uživatel v samostatném dialogu (obrázek 2 v příloze B) a to pro něj navíc velmi pohodlným způsobem, neboť jsem k tomuto účelu zvolil komponentu[56] pro práci s vlastnostmi libovolného JavaBean objektu. Tento typ komponenty jsem však zvolil záměrně, neboť jestli si vzpomínáte, v požadavcích na rozhraní (podkapitola A.8 v příloze A) jsem stanovil, že parametry Master/Worker systémů bude aplikace přijímat právě ve formě nějakého JavaBean objektu.

[55] Domovská stránka tohoto projektu se nachází pod touto URL: https://appframework.dev.java.net.

[56] Konkrétně třídu *com.l2fprod.common.propertysheet.PropertySheetPanel* z projektu L2FProd (http://www.l2fprod.com/common).

6.3.5 Druhá iterace

Ať už přesouváme data do jejich zpracovávajícího kontextu nebo kontext čili operace k datům, v Master/Worker systému vždy určité operace pracují nad nějakými daty. Ideální navíc je, pokud tento výpočet může být prováděn zcela nezávisle na ostatních, což použitý grafický engine umožňuje. Barva každého pixelu je tak počítána zcela samostatně, ovšem engine jako takový může běžet pouze v rámci jediného JVM. Není ho tedy možné spustit v gridu a mým prvním úkolem tak bude nalézt všechen kód, který je do renderování zahrnut a zároveň také data, která jsou k tomu potřeba.

Provedu tak refaktorizaci jeho stávajícího kódu, kdy některé třídy odstraním a ty zbývající rozdělím do nových balíčků. Nejdůležitější částí však bude návrh nových tříd, které budou obalovat jak data, tak operace, které jsou pro výpočet barvy jednoho pixelu potřeba.

Poté, co modifikaci enginu dokončím, budu moci konečně navrhnout rozhraní pro dorozumívání se s jednotlivými Master/Worker systémy. Toto jsem již částečně provedl v nefunkčních požadavcích a tak např. data čili scéna s hodnotami všech parametrů bude systému předávána v podobě tzv. kontextu.

Ačkoliv na konci této iterace nebude ještě žádný Master/Worker systém implementován a já tak nebudu moci přesně vědět, jaká vlákna či procesy do nich budou zahrnuta, přesto nakonec uvedu zběžný procesní pohled na celkovou architekturu vytvářené aplikace.

6.3.5.1 Výstup iterace Výstupem iterace bude refaktorovaný grafický engine. Diagram 4 v příloze C ukazuje finální rozdělení tříd do jednotlivých balíčků a reflektuje tedy jak provedenou refaktorizaci, tak existenci nových tříd. Zpravidla jsem měnil pouze názvy a formátoval zdrojový kód, ovšem třídy *Constants*, *RayTracer* a *RenderThread* jsem zcela odstranil. Těch totiž nebylo již potřeba, neboť obsahovaly jen výchozí hodnoty parametrů a nepotřebný kód pro renderování scény pouze na jednom JVM.

Většinu původních tříd jsem umístil do balíčku *raytracer.scene*, který tak např. v podobě třídy *Ray* obsahuje téměř veškerou renderovací logiku. Dále je zde také definován kompletní model scény a rozhraní abstrahující všechny její součásti. Konkrétní grafické prvky, zdroje světla a textury jsou pak obsaženy v balíčcích *raytracer.scene.texture*, *raytracer.scene.lighting* a *raytracer.scene.traceable*.

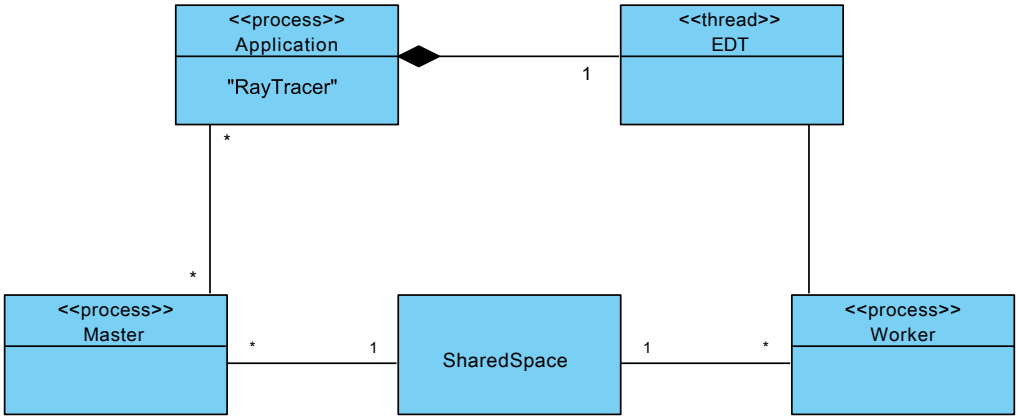
Pro výpočet barvy pixelu potřebujeme data a operace. Data zde představuje, již zmíněný, kontext, s jehož obsahem vás podrobněji seznámím až později. Operace, které na něm pak, pro získání barvy, budou vykonávat jednotliví dělníci, pro změnu implementuje třída *raytracer.scene.Ray*. Stále je však k výpočtu potřeba ještě určitý kód a k tomuto účelu jsem navrhl balíček *raytracer.engine*. Ačkoliv výhradně používám externí engine, což je ostatně dáno také nefunkčními požadavky, třídami tohoto balíku odstíním jednotlivé dělníky od konkrétních implementací. Hlavní třídou pro zahájení výpočtu je tak rozhraní *RayTracerEngine*. Kompletní obsah balíku zobrazuje diagram 5 v příloze C. Renderování iniciuje dělník vždy voláním metody *render*, jíž předá následující parametry:

- *offset* a *length* – scéna je vždy promítána na průmětnu o určité šířce a výšce. Součin těchto rozměrů pak představuje celkový počet pixelů, které je třeba vyrenderovat. A díky tomu, že sledování paprsku je ve své naivní podobě absolutně embarrassingly parallel, scénu nemusíme renderovat postupně, ale naráz po libovolně velkých částech. Takovou oblast vždy vymezíme x a y souřadnicemi jejího levého horního rohu, šířkou a výškou. Já jsem k tomuto účelu ale zvolil pouze jednu dimenzi, a tak dělníkovi pro pohyb v průmětně stačí jenom *offset* (souřadnice x) a *length* (šířka). Těmito parametry tak dělník vždy určí, jakou oblast si přeje vyrenderovat.
- *destArray* – výsledné barvy jednotlivých pixelů budou enginem umístěny do tohoto pole
- *destPos* – pole může být větší než šířka vyrenderované oblasti, a tak je nutné ještě zadat počáteční pozici

6.3.5.1.1 Rozhraní Master/Worker systému Kontext je součástí API pro komunikaci s Master/Worker systémy a definuje ho rozhraní *RayTracerContext*. Pokud se podíváte na jeho zdrojový kód (ten je dostupný na přiloženém CD), zjistíte, že obsahuje jak renderovací parametry (Recursion Level, Supersampling, Refraction Index, Diffuse Light Multiplier atd.), tak kompletní scénu (metoda *getScene*) a parametry specifické volanému systému (metoda *getConfigBean*). Tato data zůstávají po celou dobu výpočtu neměnná, a tak je není nutné mezi jednotlivými uzly sdílet. Není tak třeba žádné synchronizace a počítače mohou pracovat jen s jejich kopiemi. Embarrassingly parallel problémy tak samy o sobě již zajišťují vysokou škálovatelnost, neboť se data nemusí přesouvat po síti a kopie se mohou nechat jen v operační paměti.

Kontext a ostatní třídy, které společně vytvářejí rozhraní mezi aplikací a jednotlivými Master/Worker systémy, zobrazuje diagram 6 v příloze C. Ačkoliv název tomu příliš neodpovídá, každý systém musí implementovat rozhraní *RayTracer*. Toto jméno má však své opodstatnění, neboť aplikaci vůbec nezajímá, jaký engine požadovanou scénu (kontext) vyrenderuje, nebo zda výpočet proběhne v gridu. Využitý „RayTracer“ a Master/Worker systémy jsou totiž pouze mé nynější požadavky a v dalších verzích aplikace třeba nebudou již potřeba. Balíčky *raytracer* a *raytracer.engine* tak představují smysluplnou abstrakci grafického enginu, kdy aplikace ani netuší, že renderování probíhá v gridu nebo nějakým konkrétním enginem. Z tohoto důvodu se toto rozhraní nazývá *RayTracer* a ne např. *MasterWorker*, *DivideAndConquer* nebo *GridSystem*.

Prvky rozhraní *RayTracer* (viz zdrojový kód třídy *raytracer.RayTracer* na přiloženém CD) odpovídají požadavkům na rozhraní z podkapitoly A.8 v příloze A. Nechybí zde tedy identifikátor, srozumitelný název (*getDisplayName*), popis (*getDescription*) a volitelně ikona (*getIcon*). Dále jsou zde metody týkající se specifických parametrů a to *createConfigBean* a *getConfigBeanInfo*. Ostatní metody jsou již aktivní v procesu renderování a k jejich popisu se tak nejvíce hodí sekvenční diagram (viz obrázek 7 v příloze C). Ten tak zachycuje komunikaci aplikace s „raytracerem“, popřípadě Master/Worker systémem a znázorňuje jeho kompletní životní cyklus. Aplikace tak v něm zahájí nejenom vlastní



Obrázek 13: Procesní architektura aplikace *RayTracer*

renderování, ale počká si také na barvy všech pixelů a nakonec „raytraceru“ ukončí jeho činnost.

6.3.5.1.2 Procesní pohled na architekturu aplikace Ačkoliv jsem ještě žádný Master/Worker systém nerealizoval, a nemohu tak tudíž přesně vědět, jaká vlákna či procesy budou do výsledného systému zahrnuta, i přesto se vám nyní již pokusím vysvětlit alespoň abstraktní pohled na procesní architekturu aplikace.

Architekturu znázorňuje diagram 13. Jak můžete vidět, aplikační proces, respektive JVM, zde představuje entita *Application*. Ta bude využívat pouze jediné vlákno a to EDT. EDT již známe, a tak již víme, že se kompletně stará o uživatelské rozhraní. Vykreslování bude aplikace delegovat na „*Mastery*“, respektive *Master* na dělníky (*Worker*). Ten s nimi ovšem nebude komunikovat přímo, ale vždy prostřednictvím určitého sdíleného prostoru (*SharedSpace*). To může být jak aktivní[57], tak pasivní úložiště, ale pro nás je nyní především relevantní, že architektura aplikace se skládá ze tří hlavních procesů – *Application*, *Master* a *Worker*.

S vědomím těchto entit si můžeme konečně ukázat činnost Master/Worker systému (viz diagram 8 v příloze C), který potom, co objekt *rayTracer*, v předchozím sekvenčním diagramu (viz obrázek 7 v příloze C), obdrží zprávu *startRayTracing(RayTracerContext context)*, bude paralelně běžet se smyčkou *loop*.

6.3.6 Třetí iterace

V této iteraci navrhnu a vytvořím Master/Worker systém (pouze) pro jedno JVM. Ačkoliv tak nebude odpovídat nefunkčním požadavkům, tj. distribuovanosti, odolnosti proti chybám apod., realizuji ho z toho důvodu, abych vám na něm poté ukázal, jak málo stačí k tomu, abychom z něho, pomocí Terracotty, udělali plně distribuované, škálující, efektivní, dynamické, odolné, prostě průmyslově použitelné řešení.

[57] Například systém s rozhraním JavaSpaces (http://www.jini.org/wiki/JavaSpaces_Specification).

Systém pojmenuji *Simple RayTracer* a budu muset realizovat tato dvě rozhraní: *RayTracerProvider* a *RayTracer*. *RayTracer* již známe, ovšem prvně jmenované nikoliv. To tvoří tzv. SPI[58], kterým aplikace (dynamicky) získává různé implementace grafických enginů. Při startu programu jsou tak tito poskytovatelé automaticky vyhledáváni a prostřednictvím jejich metody *getRayTracers* si aplikace zjišťuje všechny dostupné implementace sledování paprsku.

6.3.6.1 Výstup iterace Třídní digram finálního řešení ilustruje obrázek 9 v příloze C. Ačkoliv kvůli přehlednosti jsem do něj nezakreslil zcela všechny třídy (např. *RayTracerContext*), vše důležité obsahuje. Poskytovatel enginů je zde reprezentován třídou *SimpleRayTracerProvider*, která vytváří instance *SimpleRayTracer*. *SimpleRayTracer* je tedy sledovač paprsků pro jedno JVM a implementuje tak, nám již známé rozhraní, *RayTracer*. Má tedy svůj identifikátor („*SimpleRayTracer*“), jméno („*Simple RayTracer*“), popis i ikonu. Scénu vykresluje samozřejmě po částech, a tak jako specifický parametr dále přijímá tzv. *batchSize*. Výraz *ŠÍŘKA_PRŮMĚTNY * VÝŠKA_PRŮMĚTNY / batchSize* pak určuje počet částí, respektive dávek či jednotek práce, na které se scéna rozdělí, a které se poté zašlou dělníkům. *batchSize* tak představuje počet pixelů, které bude muset dělník, v rámci jedné dávky, vždy zpracovat. K renderování však nedochází přímo v *SimpleRayTracer*, ale vše se nechává na mistrovi a dělnících. Metody *startRayTracing(RayTracerContext context)*, *stopRayTracing(RayTracerContext context)*, *awaitTermination()* a *shutdown()* jsou tak velmi jednoduché a jak ukazuje výpis D.1 v příloze D, převážně kontrolují pouze vstupní parametry. Stěžejními třídami tohoto single-JVM systému jsou tak entity mistra (*SimpleMaster*) a dělníků (*SimpleWorker*) a *SimpleRayTracer* pouze uchovává mapování renderovacích kontextů na objekty *SimpleMaster*. Toto mapování je důležité jen z toho důvodu, abychom mohli, v případě volání *stopRayTracing*, požadované renderování zastavit. Pro každý kontext je tak vždy vytvořen nový objekt *SimpleMaster*, kterého *SimpleRayTracer* spouští voláním metody *start()*. Ta je opět velice prostá a na novém vlákně pouze volá metodu *run()*. Poněvadž je *start()* zpravidla volána na EDT, spouštím mistra raději na novém vlákně, abych tak předešel blokování UI.

SimpleRayTracer tedy přenechává veškeré renderování, respektive instanci *RayTracerContext*, jenom mistrovi a ten, v metodě *run()* (viz výpis D.2 v příloze D) dle obdržené velikosti dávky, vytvoří, pro tento kontext, vždy určitý počet dělníků. V tomto Master/Worker systému tak ani mistr (*SimpleMaster*), ani dělníci (*SimpleWorker*) nepředstavují vlákna či procesy, ale pouze vlastní jednotky práce. Je tomu tak, neboť *SimpleMaster* využívá *ExecutorService*, což je JavaSE implementace Master/Worker systému, a o které jsem se již zmiňoval. S její pomocí se tak již nemusíme zaobírat vlákny, ani jejich správnou synchronizací, ale stačí nám jí pouze říct, jakou práci chceme vykonat. Jejím prostřednictvím tak mistr „vykonává“ jednotlivé dělníky, kteří poté, co svou dávku zpracují, předají získané pixely dále aplikaci. Kompletní renderovací proces tohoto řešení tak není vůbec složitý a můžete si ho prohlédnout na digramu 10 v příloze C.

ExecutorService umožňuje plánované spouštění jednotek práce a správu jejich životního cyklu (popisoval jsem jej již v kapitole 6.2.1). Toto rozhraní jsem zvolil z toho důvodu,

[58] Více na http://en.wikipedia.org/wiki/Service_Provider_Interface.

neboť Terracotta nabízí její plně distribuovanou implementaci. At'už spustíme dávku metodou *invokeAll*, *invokeAny* nebo *submit*, vždy jako návratovou hodnotu obdržíme objekt *Future*[59]. Pomocí něj řídíme životní cyklus dávky, a tak můžeme její výpočet např. zrušit, anebo si počkat na její dokončení. Objekty *Future* využívám i já a konkrétně tímto způsobem realizuji řízené ukončení renderování (viz metoda *stop* ve třídě *SimpleMaster*).

Typická implementace této služby je postavena nad, v tomto textu již zmíněným, poolem vláken a sdílenou frontou. Vlákna z poolu si vybírají z fronty stále nové a nové jednotky práce, vykonávají je a to vše až do té doby, než je služba úplně zastavena. Různé implementace nám poskytuje pomocná třída *java.util.concurrent.Executors* a já jsem konkrétně využil její metodu *newFixedThreadPool(int nThreads)*.

Příští iterace se bude konečně již zabývat Master/Worker systémem nad Teraccotou a jak uvidíte, od tohoto single-JVM řešení se nebude téměř vůbec odlišovat. Jak jsem již říkal, Terracotta nám totiž nabízí zcela distribuovanou variantu *ExecutorService* a aplikace tak nebude mít ani ponětí, že objekty *Future* ovládá dělníky, kteří třeba běží na úplně jiném počítači. Co je ovšem důležité, a jak to ostatně také ukážu, tato distribuovaná implementace se obejde zcela bez jakékoliv komunikace po síti. Terracotta si totiž vystačí pouze s POJO, jejich (low-level) sdílením a dodržováním paměťového modelu.

6.3.7 Čtvrtá iterace

V této iteraci již konečně realizuji Master/Worker systém nad Terracottou a zaměřím se na splnění všech vlastností, které jsem si vytyčil – tj. schopnost zvládat i enormní objemy dat, podpora směrování, odolnost vůči chybám jednotek práce, imunita vůči selhání dělníků a dynamická správa klastru.

6.3.7.1 Výstup iterace Zdrojový kód se oproti třetí iteraci téměř vůbec nezměnil (viz diagramy 11 a 12 v příloze C) namísto single-JVM realizace *ExecutorService* totiž využívám akorát jeho speciální distribuovanou implementaci. Tu mi Terracotta nabízí v rámci projektu TIM-Messaging, jemuž se budu věnovat více později. S pomocí tohoto projektu, tak bez jakékoliv změny zdrojového kódu, vytvořím ze single-JVM řešení plně distribuovaný systém, který bude navíc splňovat všechny výše uvedené požadavky.

Jediným rozdílem tak, mezi třetí a čtvrtou iterací, je konfigurační soubor (*tc-config.xml*), kterým pouze nadefinuji kořeny a označím určité třídy pro instrumentaci. O zbytek se již postará TIM-Messaging, virtuální halda a sdílené objekty. I tento projekt je ale založen pouze na POJO a správné *tc-config* konfiguraci, a tak veškeré složitosti distribuovaného Master/Worker systému – tj. přenos dávek a výsledků po síti, výpadky mistrů a dělníků apod. – tak skutečně zůstanou jen na Terracottě a tím hluboko pod mou aplikací včetně TIM-Messaging.

6.3.7.1.1 Konfigurační soubor tc-config.xml V konfiguračním souboru (viz podkapitola D.3 v příloze D) nejprve uvedu servery (jak již víme, pomocí elementu *servers*),

[59] Definice rozhraní viz javadoc: http://java.sun.com/javase/6/docs/api/java/util/concurrent/Future.html.

které běží či teprve budou běžet v rámci aplikačního klastru. Pro mé účely zpravidla využívám jenom aktuální počítač, a tak elementem *server* odkazuji pouze na *localhost*. Dále ponechám výchozí porty (*dso-port* a *jmx-port*), nastavím adresáře pro virtuální haldu (*data*) a také definuji složky pro žurnály (*logs*) a statistiky (*statistics*).

Následně uvedu sekci *clients*, ve které, mimo umístění klientských žurnálů, oznámím, že má aplikace (tj. jak vlastní aplikace, tak samostatný program dělníků) využívá integrační modul *tim-masterworker* (součást TIM-Messaging; více viz níže). Kdykoliv tedy spustím aplikaci nebo nového dělníka, Terracotta vždy na classpath přidá JAR tohoto modulu a také JARy, které on sám potřebuje.

V konfiguraci pokračuji výčtem tříd a rozhraní, které budou muset být instrumentovány (element *instrumented-classes*). Nejenže zde tedy vypíši třídy, které budou přímo sdíleny, ale uvedu zde samozřejmě i takové, které se sdílenými objekty pouze manipulují.

Jediným objektem, který bude v klastru přenášen, je jednotka práce (třída *raytracer.provider.terracotta.TerracottaWorkerJob*) a tak její bytecode musí být obohacen i u všech její součástí. Z tohoto důvodu zde, prostřednictvím elementů *include* a *class-expression*, uvádím implementace rozhraní *RayTracerContext* (výraz „*raytracer.RayTracerContext+*“) a také veškeré třídy, které utváří finální scénu (více viz podkapitola D.3 v příloze D).

Poslední položkou konfiguračního souboru je element *transient-fields*. Jím označím členy jinak sdílených tříd, které si nepřeji přenášet. Ať už tedy změním stav *RayTracerContextImpl.configBean* nebo *RandomMaterial.GENERATOR*, žádná z těchto modifikací tak nikdy nebude propagována do Terracotta serveru.

Možná jste si také po zhlédnutí výpisu všimli, že v konfiguraci chybí element *locks*. V aplikaci totiž nepotřebuji žádnou explicitní synchronizaci, a tak definici zámků můžu zcela vynechat. Jak později ukážu, ty deklaruje až *tim-masterworker* ve svém konfiguračním souboru a já se tak o synchronizaci nijak nestarám.

6.3.7.1.2 TIM-Messaging TIM-Messaging[60] je rozšíření Terracotty pro implementaci aplikací, které komunikují prostřednictvím výměny zpráv. Poskytuje dva integrační moduly – TIM-Pipes a TIM-MasterWorker, které mimo konfiguračních souborů definují také své vlastní API.

TIM-Pipes realizuje tzv. „lightweight messaging system“ a staví na následujících konceptech:

- Pipe neboli roura
  - Komunikační kanál podobný frontě, který mohou aplikace využít k přenosu zpráv z jednoho konce na druhý.
  - Poskytuje dva komunikační styly - point-to-point[61] a publish-subscribe[62].

[60] Domovská stránka projektu je dostupná pod touto URL: http://forge.terracotta.org/releases/projects/tim-messaging/.

[61] Přímá komunikace pouze dvou entit. Point-to-point komunikační kanál má tedy vždy jen dva konce.

[62] V publish-subscribe modelu může být zpráva doručena mnoha příjemcům. Odesílatel tak nikdy přesně neví, jakým a kolika příjemcům, bude zpráva aktuálně doručena.

- Topology neboli topologie
  - Je hlavním přístupovým bodem ke všem rourám konkrétního „messaging“ systému.
  - Definuje kongregaci rour, ve které je každá roura identifikována tzv. směrovacím identifikátorem.
- Router neboli směrovač
  - Zatímco k rourám přistupujeme pomocí jejich směrovacího identifikátoru, dle jeho hodnoty mohou být roury také vybírány. Tento koncept realizuje právě *Router* a snaží se tak vždy, dle určitého směrovacího algoritmu, vybrat takovou rouru, která bude reprezentovat aktuálně nejlepší cestu v rámci klastru k určité jeho entitě.

TIM-MasterWorker využívá roury pro implementaci jednoduchého, ale zároveň škálovatelného Master/Worker systému. Ten umožňuje zasílání dávek z jednoho nebo více mistrů na libovolný počet distribuovaných dělníků a poskytuje hned dvě implementace tohoto architektonického vzoru:

- *WorkManager*
  - *WorkManager* je založen na specifikaci CommonJ Work Manager a realizuje její čistě „POJO-based“ implementaci.
- *ExecutorService*
  - Realizuje nám již známé rozhraní *java.util.concurrent.ExecutorService* a umožňuje tak spouštění *Runnable*[63] a *Callable*[64] na distribuovaných dělnících.

Každý Master/Worker systém postavený nad TIM-MasterWorker sdružuje své mistry a dělníky vždy v rámci jedné topologie. Ta přiřadí každému mistrovi a dělníku rouru s určitým směrovacím ID a umožní jim tak navzájem komunikovat. Jak již víte, pro potřeby mého projektu jsem si zvolil implementaci *ExecutorService*, ale i kdybych býval zvolil *WorkManager*, TIM-MasterWorker mi zaručí vždy následující chování:

- Mistři a dělníci sdílející stejnou topologii, jsou k topologii vždy automaticky připojeni a ihned tak vidí i všechny ostatní.
- Poté, co prostřednictvím určitého mistra odešleme jednotku práce, mistr na základě asociovaného *Routeru* zvolí vždy nejlepší trasu k jednomu z běžících dělníků.

[63] Rozhraní, které musí implementovat všechny Java třídy, jejichž instance chtějí být vykonány na externím vlákně, respektive třídou *java.lang.Thread*.

[64] Rozhraní, které realizuje stejný účel jako rozhraní *Runnable*, ale může vyhazovat kontrolovanou výjimku a vracet výsledek.

  - Pokud žádná taková trasa neexistuje, dávka je topologií uložena do tzv. fronty neodeslaných jednotek. Ta je vyprazdňována tehdy, pokud se připojí nový dělník, anebo když jakýkoliv z dělníků mistrovi oznámí, že je připraven vykonat novou práci.
  - Pokud nějaká cesta nalezena je, dávka se uloží do tzv. fronty odeslaných jednotek a poté se rourou odešle vybranému dělníku.

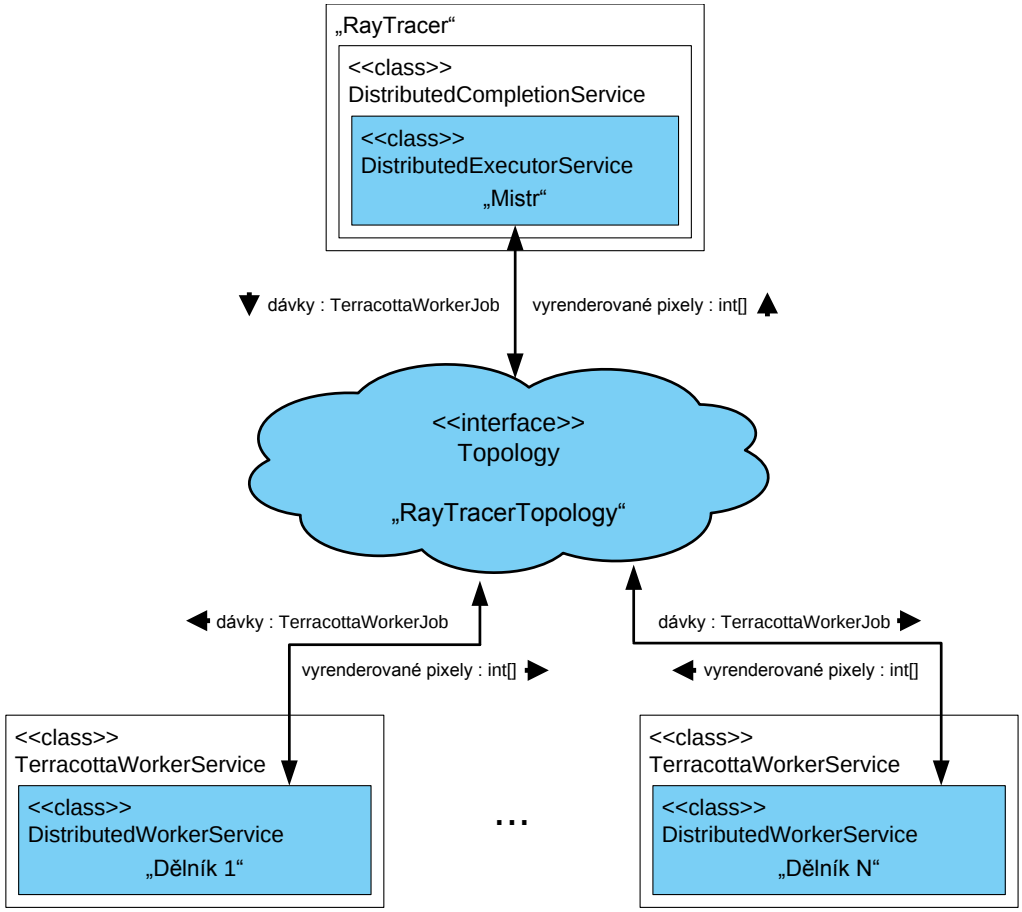
- Dělníci si vždy vybírají dávky ze své roury, vykonávají je a výsledky zasílají zpět mistrovi, který danou práci odeslal.
- Kdykoliv mistr obdrží od dělníka dokončenou práci, tak tuto jednotku odstraní z fronty odeslaných jednotek.
- Pokud se mistr odpojí, anebo selže a v topologii běží alespoň ještě jeden další mistr, tak všechny jednotky práce (odeslané i neodeslané) tohoto mistra budou jednomu z běžících mistrů přiděleny. Ten je pak regulérně odešle dělníkům, jako by to udělal ten odpojivší mistr.
- Stejně tak když selže dělník, anebo se odpojí, všechna jeho práce bude daným mistrem přeposlána jinému dělníku.

TIM-MasterWorker, respektive TIM-Messaging splňuje všechny požadavky kladené na průmyslově použitelný Master/Worker systém:

- Každý dělník i mistr má svou vlastní rouru, a tak nedochází k přílišnému soupeření. Systém je proto schopen zvládat i enormní objemy dat.
- Více rour si již vyžaduje směrování a třída Router toto přesně splňuje.
- Odolnost vůči chybám mistrů a dělníků jsem zmínil před chvílí, pouze selhání jednotky práce si musí programátor řešit sám.
- Dynamickou správu klastru nám poskytuje topologie, neboť umožňuje mistrům a dělníkům se do ní kdykoliv připojit, anebo se z ní odpojit.

6.3.7.1.3 RayTracerProvider pomocí TIM-MasterWorker V třídním (viz obrázek 11 v příloze C), respektive sekvenčním (obrázek 12 v příloze C) diagramu jsem ukázal, že jediné, co moje aplikace musí pro získání plně distribuovaného Master/Worker systému udělat, je využít pouhé dvě třídy – *DistributedExecutorService* a *DistributedWorkerService*. Ty reprezentují mistra a dělníka a já díky nim, oproti třetí iteraci, opravdu nemusím nijak měnit svůj zdrojový kód. Aplikace tedy vůbec nevidí tu složitou komunikační logiku, synchronizaci, cachování a další všemožné aspekty, které jsou pro fungování klastru potřeba. A to je právě to, o co Terracottě jde, čili co možná nejvíc vývoj distribuovaných systémů programátorům usnadnit.

Architekturu poskytovatele grafického enginu postaveného nad Terracottou, respektive využití TIM-MasterWorker modulu v mé aplikaci, ilustruje obrázek 14. Po spuštění



Obrázek 14: Architektura grafického enginu postaveného nad Terracottou

aplikace se „RayTracer“ ihned spojuje s Terracotta serverem a poté, co uživatel zažádá o vyrenderování scény prostřednictvím tohoto poskytovatele (v programu jako „*Terracotta RayTracer*“), aplikace se pomocí *DistributedExecutorService* připojí do již existující topologie („*RayTracerTopology*“). Mistr tedy běží v rámci virtuálního stroje aplikace a ihned si, po přijetí renderovacího kontextu, scénu rozdělí na určitý počet jednotek (*TerracottaWorkerJob*). Ty pak dle určitého směrovacího algoritmu odešle prostřednictvím jejich rour vybraným dělníkům – tj. instancím *DistributedWorkerService*, které jsou spouštěny mým dedikovaným programem (*TerracottaWorkerService*). Ty pak přijaté dávky zpracují a vyrenderované pixely, jako celočíselná pole, odesílají zpět mistrovi. Ten pro ně následně vytváří události typu *ScreenUpdateEvent*, kterými nakonec notifikuje aplikaci. A tento proces běží tak dlouho, dokud mistr neobdrží i ty poslední pixely.

Pokud tedy já nemusím řešit složitosti distribuovaného programování, kdo tedy? Ano, TIM-MasterWorker sice pro mne kompletně implementuje vysoce výkonný Master/Worker systém, ale, jak už jsem řekl, i on je složen pouze z relativně jednoduchých POJO.

Veškeré nízko-úrovňové aspekty tak opravdu zůstávají jenom na bedrech Terracotty a na tomto projektu je tak krásně vidět, jak i průmyslově použitelného řešení můžeme docílit pouhým sdílením objektů. Vše je tedy pro mou aplikaci a TIM-Messaging zcela transparentní a data se tak v topologii opravdu přesouvají pouze na úrovni JVM.

Podrobný popis fungování TIM-Messaging, respektive TIM-MasterWorker je uveden v příloze F.

6.3.8 Pátá iterace

Úkolem této iterace je realizovat Master/Worker systém, respektive implementovat nového poskytovatele grafického enginu, bez použití Terracotty. Řešení musí být navíc implementováno tak, aby splňovalo, pokud možno, opět všechny vytyčené vlastnosti – tedy být schopno vykonávat spoustu dávek najednou, provádět load balancing, být rezistentní vůči chybám dělníků a umožňovat dynamickou správu klastru.

6.3.8.1 Výstup iterace K řešení jsem využil projekt JPPF[65], který je průmyslovou implementací Master/Worker systému. Zvolil jsem jej z toho důvodu, neboť splňuje všechny výše požadované vlastnosti a je pěkně navržen.

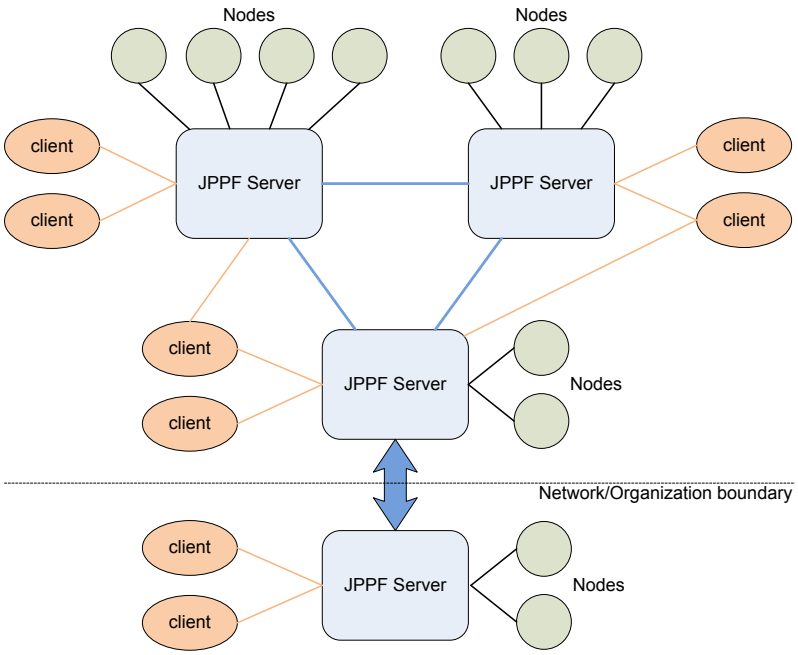
Stejně jako ve čtvrté iteraci, i zde začněme popisem architektury. JPPF není framework pro tvorbu libovolných distribuovaných systémů, ale implementuje pouze vzor Master/Worker. Jeho architektura je tedy velmi jednoduchá a skládá se pouze ze tří základních entit. Jak můžete vidět na obrázku 15, těmito entitami jsou *client*, *JPPF Server* a *Node*. Uzly *client* zde reprezentují aplikace, *JPPF Server* mistry a *Nodes* jednotlivé dělníky. Tyto entity mohou vytvářet libovolně provázanou síť a stejně jako u Terracotty, i zde, pokud např. selže jeden z mistrů či dělníků, ostatní uzly ihned přeberou jeho nedokončenou práci.

Toto tedy byla architektura z „ptačí“ perspektivy a nyní se podíváme, jakým způsobem jsou zpracovávány požadavky jednotlivých klientů. Princip zpracování dávky zobrazuje obrázek 16. Jak můžete vidět, dávky jsou odesílány metodou Submit, která je součástí klientského API, jež má každý klient k dispozici. Tato metoda dávku odešle na požadovaný server, který si ji uloží do své fronty. Odtud si ji následně vyzvedne subsystému s názvem *Task Bundler*, který je v JPPF zodpovědný za směrování, respektive load balancing. Ten již provádí odesílání dávek na jednotlivé dělníky. Vždy, když dělník (*Node*) dávku přijme (ta se v JPPF nazývá *Task*), provede nejprve její dekódování (tj. také deserializaci), poté ji spustí a nakonec odešle její výsledky.

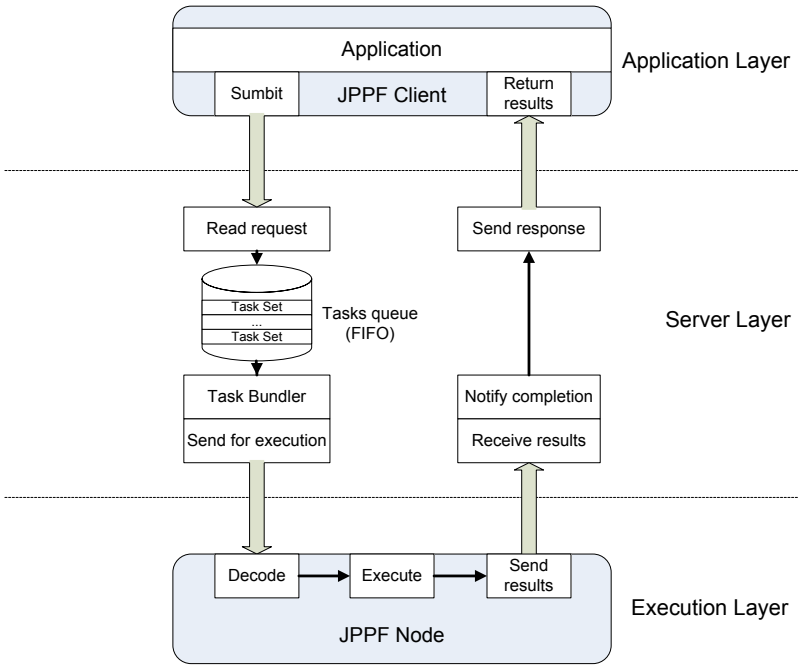
Teď již k samotnému řešení. K jeho popisu jsem zvolil třídní diagram (viz obrázek 15 v příloze C), který sice vypadá velký, ale složitý není. Velký je především z toho důvodu, neboť obsahuje také třídy a rozhraní ze společného API pro poskytovatele grafických enginů. Tyto třídy zde již nebudu popisovat, ale uvedl jsem je proto, abych ukázal, jakým způsobem je JPPF řešení do aplikace *RayTracer* integrováno.

Kdykoliv chce aplikace vyrenderovat scénu pomocí JPPF poskytovatele, vytvoří se pro daný kontext (*RayTracerContext*) objekt typu *JPPFMaster*. Ten, stejně jako minulá řešení,

[65] Tento projekt je dostupný z této adresy: http://www.jppf.org.



Obrázek 15: Architektura projektu *JPPF*



Obrázek 16: Princip zpracování dávek v projektu *JPPF*

scénu nejprve rozdělí na určitý počet částí a pro každou tuto část vytvoří objekt typu *RayTracerJPPFTask*. Tento objekt rozšiřuje třídu *JPPFTask*, která v projektu JPPF reprezentuje dávku, a může tak být vykonávána JPPF dělníky. Dávky avšak nemohou být odesílány mistrovi samostatně, ale vždy v rámci tzv. „jobů“, respektive objektů *JPPFJob*. *JPPFJob* tedy slouží k sdružení všech dávek, které patří k jednomu výpočtu a v mém případě je to vyrenderování jedné scény. *JPPFMaster* tedy po vytvoření všech dávek následně vytvoří také jeden *JPPFJob*, kterému poté, metodou *addTask(JPPFTask)*, všechny dávky předá. Aby mohlo vyrenderování scény proběhnout, musím *JPPFJob* nejprve nakonfigurovat. Metodou *setId(String)* mu nastavuji jednoznačné ID, které potřebuji pro jeho případné zastavení. Metodu *setDataProvider(DataProvider)* volám z toho důvodu, aby měly jednotlivé dávky, respektive tasky, během vykonávání na dělnících, přístup k renderovacímu kontextu. Další metodou, kterou je nutné zavolat, je *setResultListner(TaskResultListener)*. Tato metoda nastavuje posluchače, kterému jsou zasílány ty tasky, které již dokončili svůj výpočet. *JPPFMaster* toto rozhraní implementuje, a tímto způsobem tak získává vyrenderované pixely jednotlivých dávek.

Zbývá popsat třídy *JPPFClient* a *JMXDriverConnectionWrapper*. Jak asi tušíte, pomocí třídy *JPPFClient* se aplikace *RayTracer* připojuje k JPPF Serverům. JPPF Servery jsou samozřejmě spouštěny mimo aplikaci a při jejím startu již musí běžet. Poněvadž *JPPFClient* je jedinou třídou, která komunikuje se serverem, obsahuje metodu *submit(JPPFJob)*, kterou se odesílá „renderovací“ job. Jelikož *RayTracer* také umožňuje renderování ukončit, JPPF obsahuje také třídu *JMXDriverConnectionWrapper* s metodou *cancelJob(String jobId)*. Kdykoliv je tato metoda zavolána, je job s daným *jobId* ukončen.

7 Výkonnostní porovnání jednotlivých řešení

Tato kapitola je věnována výkonnostnímu porovnání Master/Worker systémů Terracotta a JPPF. Jejich výkon si ověřím na testovací scéně, přičemž budu sledovat, jak efektivně dokážou využít dostupný hardware. Otestuji tedy jejich škálovatelnost, kdy pro výpočet nejprve vyhradím pouze jedno jádro, poté dvě, tři a nakonec celkově čtyři. Získané výsledky, respektive časy potřebné k vyrenderování scény, také následně okomentuji a pokusím se vysvětlit, proč byly naměřeny právě tyto výsledky.

Pro výkonnostní porovnání jsem si zvolil testovací scénu o rozměrech 800x600 pixelů s 20 světelnými objekty a 75 koulemi. Testoval jsem na rodinných osobních počítačích, kterých máme doma celkově tři. Jedná se o jeden notebook a dva stolní počítače. Jejich detailní konfiguraci si můžete prohlédnout v podkapitole E.1 v příloze E, zde bych chtěl pouze popsat model nasazení pro testovací potřeby (viz podkapitola E.2 v příloze E).

Rozhodl jsem se, že notebook bude plnit pouze roli serverů a stolní počítače pak roli dělníků. Na notebook jsem tedy umístil jak Terracotta server, server JPPF, tak i vlastní aplikaci. Na zbylých dvou počítačích jsem poté spouštěl jednotlivé dělníky a to nejprve jednoho, dva, potom tři a nakonec čtyři. Tento počet jsem zvolil záměrně a to s ohledem na dostupná výpočetní jádra. Každý počítač jich má totiž jenom dvě a na každém jsem tak mohl rozběhnout maximálně dva dělníky (dva pro každou z technologií).

Tím, že jsem ihned netestoval pouze na všech čtyřech jádrech, ale postupně, jsem získal možnost ověřit si škálovatelnost obou systémů. Zajímalo mne tedy, zda škála, s rostoucím počtem jader, bude u systémů lineární, anebo zda, na výsledných časech renderování, bude zřetelná systémová režie.

Oba systémy podporují dávkové zpracování, a tak jsem nejprve provedl testy s velikosti dávky 5000 pixelů, poté s 1000 pixely a nakonec pouze 500 pixely. S ohledem na rozměry scény (800x600=480000 pixelů) tak musely, v prvním případě, systémy zpracovat 96 dávek, v druhém 480 a nakonec ve třetím 960 dávek.

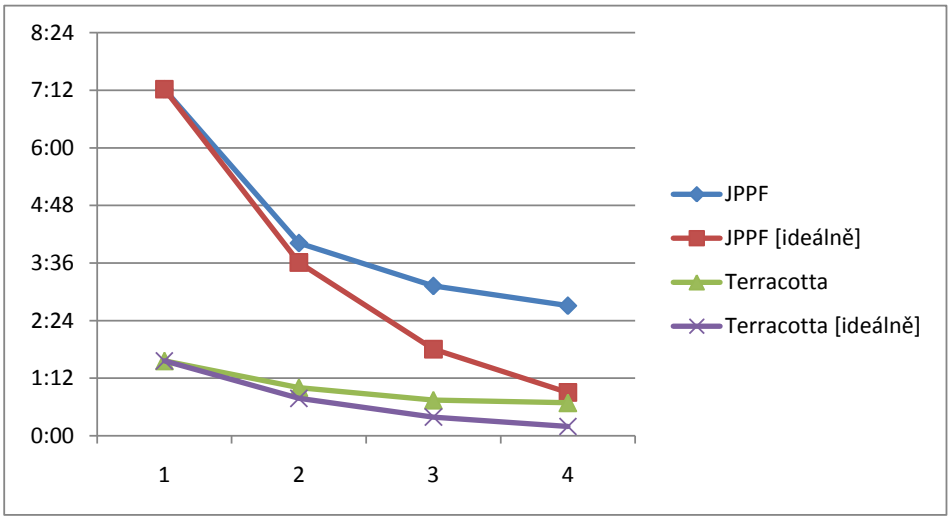
Nastavení systémů, respektive mistrů a dělníků, jsem ponechal téměř výchozí, pouze jsem nastavil, at' program dělníků využívá vždy pouze jedno vlákno. Dále jsem zajistil, at' výsledky jsou aplikaci posílány ihned a ne dávkově – to z toho důvodu, aby byla scéna vykreslována plynule a ne skokově, tj. např. po 50 dávkách.

Finální výsledky ukazuje tabulka E.3 v příloze E. Scénu jsem, pro přesnost, při daném nastavení, vyrenderoval vždy třikrát a následně jsem ze získaných časů udělal průměr. Tyto průměrné časy jsem poté vizualizoval pomocí spojnicových grafů, kterých je, podle velikosti dávky, celkově tři. Každý graf tedy ilustruje, jak si systémy průměrně vedly při dané velikosti dávky a s daným počtem dělníků. Do grafu jsem také, pro každý systém, zakreslil tzv. „ideální" křivku, která vždy začíná v dosaženém čase s jedním dělníkem a poté, s každým novým dělníkem, jsem čas dvakrát zmenšil. Tato křivka tak představuje lineární škálu, kterou by systém, v ideálním případě, mohl dosáhnout. Předpokládal jsem, že čas by měl být zhruba dvakrát tak menší, pokud vždy zdvojnásobím[66] výpočetní výkon. Grafy jsou zobrazeny v následující podkapitole.

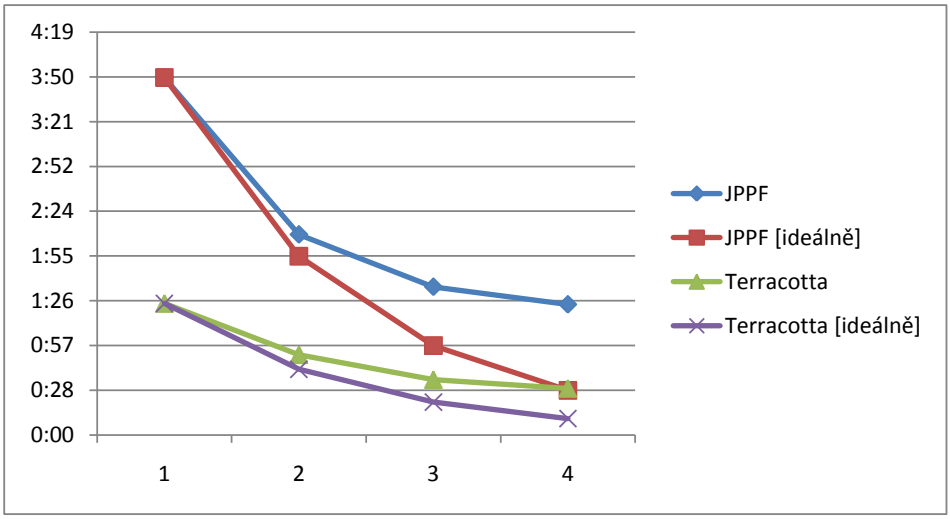
[66] Zhruba, protože počítače nemají stejný výkon.

Výsledky ukázaly, že ani jeden ze systémů se nedokázal přiblížit ideální hranici, a že tedy značná část výpočetního výkonu padla na systémovou režii – zpracování dat, serializace, synchronizace apod. Při velikosti dávky 5000 pixelů si byly systémy ještě vcelku rovny (JPPF o trochu rychlejší), ovšem s nárůstem počtu dávek se situace rapidně změnila. Ukázalo se, že čím více dávek musí systém zpracovat, tím je Terracotta podstatně rychlejší.

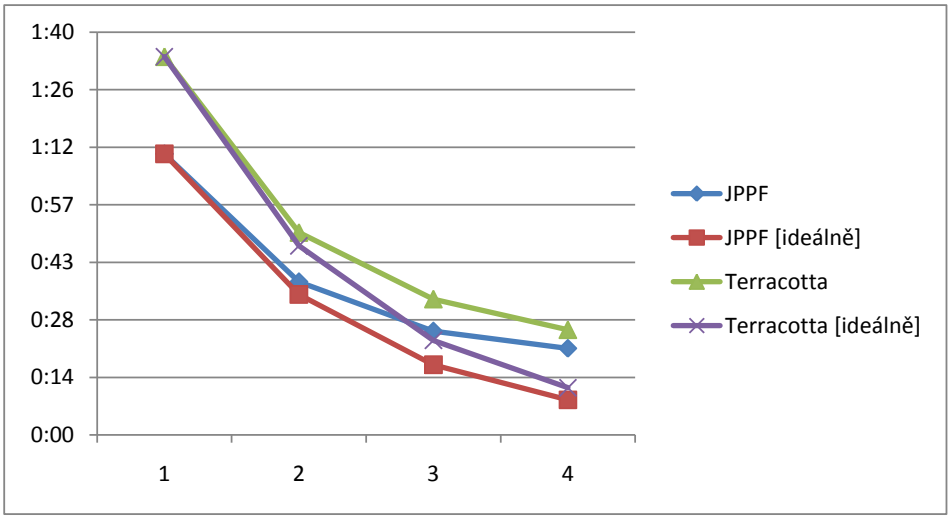
7.1 Vizualizace naměřených výsledků



Obrázek 17: Graf výsledků pro dávku o velikosti 500 pixelů



Obrázek 18: Graf výsledků pro dávku o velikosti 1000 pixelů



Obrázek 19: Graf výsledků pro dávku o velikosti 5000 pixelů

8 Převod existující aplikace na platformu Terracotta

V této kapitole bych chtěl podat návod, jakým způsobem lze již existující distribuovanou aplikaci portovat na platformu Terracotta.

Předtím, než ale začneme o převodu aplikace vůbec přemýšlet, je dobré si uvědomit, co nám Terracotta vlastně nabízí. Převod by jinak samozřejmě neměl smysl, pokud by Terracotta neměla aplikaci co nabídnout. Připomeňme si tedy definici Terracotty: Terracotta je transparentní klastrovací služba pro Java aplikace. Je to technologie, která klastrování přenáší z architektury do aplikační infrastruktury. Nabízí tedy především snadný způsob, jak lze, bez jakéhokoliv složitého kódu a databáze, transparentně škálovat a zároveň zachovat vysokou dostupnost.

Terracotta je ze své podstaty předurčena k opravdu širokému použití. Pro mnohé typy aplikací tak nabízí takové možnosti, které s výhodou mohou využít. Nejrychlejší cestou, jak lze Terracottu integrovat do stávajících řešení, jsou integrační moduly (viz podkapitola 5.2.2). Těch totiž existuje spoustu a většinou každý typ aplikace najde takový modul, který pomůže v řešení nějakého jejího problému. Důležité je, že, s integračními moduly, využití Terracotty aplikaci zpravidla nic nestojí.

Příkladem mohou být např. webové aplikace, kterým Terracotta nabízí „out-of-the-box“ sdílení relací. To realizuje prostřednictvím integračních modulů, které jsou již vytvořeny pro konkrétní aplikační servery (např. pro Jetty, Tomcat, Glassfish nebo JBoss AS). Pokud se tedy některý z těchto aplikačních serverů spustí, s daným modulem, nad Terracottou, budou automaticky sdíleny relace všech jeho aplikací. Toto vše bude samozřejmě pro aplikace naprosto transparentní.

Z důvodu, že typů distribuovaných aplikací a integračních modulů je spousta, nebudu zde popisovat různé druhy převodů, ale raději uvedu pouze obecně použitelná pravidla, která umožní převod typické distribuované aplikace na platformu Terracotta.

8.1 Algoritmus převodu

V zásadě lze říci, že pokud distribuovaná aplikace sdílí, mezi svými počítači, nějaký stav, lze ji portovat do Terracotty. S výhodou totiž můžeme využít transparentní distribuovanou cache, respektive automatické sdílení objektů, které nám Terracotta nabízí.

Co tedy musíme udělat proto, abychom aplikaci převedli? Na tuto otázku se pokusím odpovědět příkladem. Konkrétně nám známou aplikací *HelloClusteredWorld* z kapitoly 5.1. Představte si, že původní program *HelloClusteredWorld* jste již na Terracottu převedli a že výpis 2 tak ukazuje zdrojový kód po převodu. Jak ale vypadal program předtím a co jsme museli pro jeho převod udělat?

Program mohl vypadat asi tak, jak ukazuje výpis 13. Vidíme, že na rozdíl od Terracotty je zde potřeba nějakého externího API. Konkrétně metody *get(String key)* a *put(String key, Object value)*, kterými programátor získává a načítá sdílené objekty z cache. Pokud zavolá *put*, může kopii value později získat voláním *get*.

```
public class HelloClusteredWorldWithAnAPI {
    ...
```

```
    private static char[] buffer;

    ...

    public static void main( String args[] ) throws Exception {
        while( true ) {
            // explicitni synchronizace zde jiz neni potreba
            // synchronized( buffer ) {

            // ziskani aktualni hodnoty sdileneho bufferu
            // poznamka: pravdepodobne dojde k deserializaci
            buffer = cluster.get( "HelloClusteredWorldBufferID" );

            // prace s bufferem ...

            // poznamka: pravdepodobne dojde k serializaci
            Cluster.put( "HelloClusteredWorldBufferID", buffer );
            Thread.sleep( 100 );
            // }
        }
    }
}
```

Výpis 13: Distriubovaná varianta *HelloClusteredWorld* bez použití Terracotty

Oproti řešení s Terracottou zde již není potřeba synchronizace a aplikace provádí serializaci a deserializaci vždy celého bufferu do/ze sítě.

Ať už *get* či *put* nahradíme jinou technologií[67], kód po převodu bude vždy stejný. Na něm tedy krásně vidíme, co vše za nás Terracotta provádí. Porovnáním zdrojových kódů dojdeme k závěru, co je zhruba potřeba udělat, abychom obecnou aplikaci portovali do Terracotty:

1. Nejprve musíme identifikovat sdílené objekty čili data, která jednotlivé aplikační uzly mezi sebou sdílejí.
2. Nyní pro tyto sdílené objekty vytvoříme kořeny. Buď tedy sdílené objekty označíme přímo jako kořeny, anebo budeme sdílet kolekce, do kterých budeme tyto objekty ukládat.
3. Kolem kořenů vytvoříme libovolné API. Tímto API budeme nejenom sdílené objekty vytvářet (tím, že je přidáme do hierarchie jednoho z kořenů), ale také, dle potřeby aplikace, s nimi libovolně manipulovat.
4. Dále musíme ve zdrojovém kódu identifikovat ta místa, kde se sdílenými objekty pracujeme. Poté tato místa, respektive jejich stávající kód, nahradit za API, které jsme si vytvořili v kroku 3.

[67] Například technologiemi jako je JDBC, JMS nebo RMI. Metoda *put* by zde byla nahrazena voláním metod *statement.execute(UPDATE)*, *producer.send()*, respektive *objectImpl.invoke()*. Metoda *get* pak například metodami *statement.execute(SELECT)*, *consumer.receive()* a *objectImpl.request()*.

5. Integraci nového API do původního kódu máme hotovou a nyní se tedy můžeme pustit do synchronizace. Platí, že v synchronizovaných blocích, respektive v kontextech zámků bychom měli strávit, co nejméně času. Toho dosáhneme tím, že aplikujeme tzv. „fine-grained“ přístup, kdy zámky a synchronizované bloky umístíme opravdu až tam, kde už je to nezbytně potřeba. V našem případě to znamená vložení synchronizace přímo do API z kroku 3 a ne do míst, které jsme nalezli v kroku předchozím. Výjimku uděláme pouze tehdy, pokud kód provádí mnoho změn najednou a tyto změny chce provést v jedné jediné transakci. Tehdy totiž využijeme přístup zcela opačný, tzv. „coarse-grained“, kdy do převedeného kódu, a ne tedy do našeho API, umístíme explicitní synchronizaci a v kontextu pouze jednoho zámku provedeme všechny požadované změny do sdílených objektů. Explicitní synchronizaci, do převedeného kódu, dáme také tam, kdy přímo, tj. bez API z kroku 3, přistupujeme ke sdílenému objektu. Zde mám na mysli především přímý přístup ke kořenům.

6. Na základě kořenů, vytvořeného API a převedeného kódu sestavíme konfigurační soubor *tc-config.xml*. Kořeny tedy definujeme elementem *roots*, dále vytvoříme seznam *instrumented-classes* a nakonec elementem *locks* definujeme synchronizaci. U synchronizace navíc, dle konkrétní metody čili tam, kde v kódu s určitým sdíleným objektem manipulujeme, provedeme taky rozhodnutí, zda zámek bude sdílený (zámek typu *read*), anebo exkluzivní (zámek typu *write*).

7. Do *tc-config.xml* doplníme ostatní údaje: servery (*servers*), klienty (*clients*), *transient-fields*, distribuované metody (*distributed-methods*) atd.

9 Závěr

Cílem této práce bylo popsat platformu Terracotta, navrhnout vlastní řešení dle jejího konceptu a toto řešení následně nad ní implementovat. Poté tuto aplikaci realizovat také bez její pomoci a nakonec tato dvě řešení vzájemně porovnat. Posledním úkolem pak bylo podat obecný návod, jakým lze libovolnou distribuovanou aplikaci převést na tuto platformu.

9.1 Přínos této práce

Všechny cíle byly splněny. Domnívám se, že platformu Terracotta jsem popsal opravdu zevrubně, neboť jsem ji věnoval celé tři kapitoly. Nejdříve jsem uvedl její historii a nastínil důvody, které vedly k jejímu vzniku. Následně jsem velmi podrobně popsal její definici a to na její analogii k NAS. Nakonec jsem, formou jednoduchého příkladu, čtenáři ukázal, jak single-JVM aplikaci lze nad Terracottou jednoduše distribuovat. A aby byl popis Terracotty kompletní, popsal jsem také téměř všechny elementy jejího konfiguračního souboru. Myslím si, že přínosem těchto kapitol je poměrně ucelený popis Terracotty a její přiblížení českým programátorům. Prozatím jsem totiž neviděl, že by se někdo, ve své závěrečné práci, Terracottou zabýval.

Jako vlastní řešení jsem si zvolil implementaci architektonického vzoru Master/Worker. Konkrétně jsem nad ním realizoval program renderující 3D scény pomocí algoritmu sledování paprsku. Realizaci programu jsem rozdělil do několika vývojových iterací. V první iteraci jsem implementoval uživatelské rozhraní programu a to prozatím bez jakékoliv aplikační logiky. Tu jsem postupně přidával v následujících iteracích, než aplikace dosáhla takového stavu, který jsem si stanovil ve vizi. Po páté iteraci tedy byla aplikace schopna renderovat scénu jak s Terracottou, tak bez ní a to konkrétně pomocí projektu JPPF. Obě řešení jsem následně také porovnal a výsledky srovnání umístil do kapitoly 7. Přínosem této části práce je ukázka využití Terracotty při řešení reálného problému.

V poslední kapitole jsem uvedl obecně použitelná pravidla, která programátoři, již existujících, distribuovaných aplikací mohou použít, pokud své aplikace chtějí portovat na platformu Terracotta.

9.2 Další vývoj

Ačkoliv jsem čtenáři ukázal praktické využití Terracotty na Master/Worker systému, chtěl bych ji také otestovat na jiném, než čistě embarrassingly parallel problému. Zajímala by mne totiž její škálovatelnost a tím i výkonnost, pokud by se často měnila sdílená data.

Také bych se chtěl více věnovat hlavnímu zaměření Terracotty, kterým jsou webové aplikace. Popsal bych relevantní integrační moduly, přičemž bych se zaměřil především na její spojení s technologiemi Ehcache, Hibernate a Spring.

Dalším plánem je udělat zevrubnější testování. Do měření bych chtěl totiž zapojit i profilování, abych přesně zjistil, zda to, co má největší (negativní) vliv na výsledný výkon je komunikace po síti, synchronizace anebo něco jiného.

Čtenářům bych chtěl nakonec také představit tzv. Terracotta Developer Console, což je monitorovací a diagnostický nástroj pro Terracottu.

10 Literatura

[1] ZILKA, Ari; HARTLEY, Jeff. *The Definitive Guide to Terracotta : Cluster the JVM™ for Spring, Hibernate, and POJO Scalability*. First edition. United States of America : Apress, Inc., 2008. 368 s. ISBN 978-1-59059-986-0.

[2] TANENBAUM, Andrew S.; VAN STEEN, Maarten. *Distributed systems : Principles and paradigms*. Second edition. Upper Saddle River, NJ 07458 : Pearson Education, Inc., 2007. 704 s. ISBN 0-13-239227-5.

[3] COCKBURN, Alistair. *Use Cases : Jak efektivně modelovat aplikace*. Vydání první. Brno : CP Books, a.s., 2005. 262 s. ISBN 80-251-0721-3.

11 Přílohy

A Funkční a nefunkční požadavky aplikace RayTracer

- Příloha umístěna na CD v souboru *diplomova_prace_prilohy.pdf*.

B Uživatelské rozhraní aplikace RayTracer

- Příloha umístěna na CD v souboru *diplomova_prace_prilohy.pdf*.

C Diagramy aplikace RayTracer

- Příloha umístěna na CD v souboru *diplomova_prace_prilohy.pdf*.

D Výpisy hlavních zdrojových kódů aplikace RayTracer

- Příloha umístěna na CD v souboru *diplomova_prace_prilohy.pdf*.

E Detaily k výkonnostnímu porovnání jednotlivých řešení

- Příloha umístěna na CD v souboru *diplomova_prace_prilohy.pdf*.

F Integrační modul TIM-MasterWorker

- Příloha umístěna na CD v souboru *diplomova_prace_prilohy.pdf*.

G Ukázková aplikace *HelloClusteredWorld*

- Aplikace umístěna na CD ve složce *HelloClusteredWorld*.

H Aplikace *RayTracer*

- Aplikace umístěna na CD ve složce *RayTracer*.